ARM® Architecture Reference Manual

ARM®v7-A and ARM®v7-R edition

ARM®

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.
ARM DDI 0406B

ARM® アーキテクチャ リファレンスマニュアル

ARM®v7-A および ARM®v7-R エディション

抄訳

ARM®

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.
ARM DDI 0406B

ARM Architecture Reference Manual
ARMv7-A and ARMv7-R edition

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Confidentiality	Change
05 April 2007	A	Non-Confidential	New edition for ARMv7-A and ARMv7-R architecture profiles. Document number changed from ARM DDI 0100 to ARM DDI 0406 and contents restructured.
29 April 2008	B	Non-Confidential	Addition of the VFP Half-precision and Multiprocessing Extensions, and many clarifications and enhancements.

From ARMv7, the ARM® architecture defines different architectural profiles and this edition of this manual describes only the A and R profiles. For details of the documentation of the ARMv7-M profile see *Further reading* on page xx. Before ARMv7 there was only a single *ARM Architecture Reference Manual*, with document number DDI 0100. The first issue of this was in February 1996, and the final issue, Issue I, was in July 2005. For more information see *Further reading* on page xx.

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

1. Subject to the provisions set out below, ARM hereby grants to you a perpetual, non-exclusive, nontransferable, royalty free, worldwide licence to use this ARM Architecture Reference Manual for the purposes of developing; (i) software applications or operating systems which are targeted to run on microprocessor cores distributed under licence from ARM; (ii) tools which are designed to develop software programs which are targeted to run on microprocessor cores distributed under licence from ARM; (iii) or having developed integrated circuits which incorporate a microprocessor core manufactured under licence from ARM.

2. Except as expressly licensed in Clause 1 you acquire no right, title or interest in the ARM Architecture Reference Manual, or any Intellectual Property therein. In no event shall the licences granted in Clause 1, be construed as granting you expressly or by implication, estoppel or otherwise, licences to any ARM technology other than the ARM Architecture Reference Manual. The licence grant in Clause 1 expressly excludes any rights for you to use or take into use any ARM patents. No right is granted to you under the provisions of Clause 1 to; (i) use the ARM Architecture Reference Manual for the purposes of developing or having developed microprocessor cores or models thereof which are compatible in whole or part with either or both the instructions or programmers' models described in this ARM Architecture Reference Manual; or (ii) develop or have developed models of any microprocessor cores designed by or for ARM; or (iii) distribute

抄訳

ARM アーキテクチャ リファレンスマニュアル
ARMv7-A および ARMv7-R エディション

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

リリース情報

本書には、次の変更が加えられています。

改訂履歴

日付	変更箇所	公開の有無	変更内容
2007 年 4 月 5 日	A	公開	ARMv7-A および ARMv7-R アーキテクチャプロファイル用の新規エディション。 ドキュメント番号が ARM DDI 0100 から ARM DDI 0406 に変更され、内容が再構成されました。
2008 年 4 月 29 日	B	公開	VFP 半精度およびマルチプロセッシング拡張機能の記述を追加し、各所の記述の明確化と追記を行いました。

ARMv7 から、ARM® アーキテクチャは各種のアーキテクチャプロファイルを定義するようになりました。本書のこのエディションでは、A および R プロファイルについてのみ解説しています。ARMv7-M プロファイルの詳細なドキュメントについては、*参考資料* P. xx を参照して下さい。ARMv7 以前の『*ARM アーキテクチャ リファレンスマニュアル*』は 1 つだけで、資料番号は DDI 0100 です。このマニュアルの初版は 1996 年 2 月に発行され、最終版であるリリース I は 2005 年 7 月に発行されました。詳細については、*参考資料* P. xx を参照して下さい。

著作権表記

® または ™ の付いた用語とロゴは、本著作権条項で特に明記されていない限り、EU および他諸国における ARM Limited の登録商標または商標です。本書に記載されている他の商標その他の名前は、対応する所有者の商標の場合があります。

本書に記載されている情報の全部または一部、ならびに本書で紹介する製品は、著作権所有者の文書による事前の許可を得ない限り、転用・複製することを禁じます。

本書に説明されている製品は、継続的に開発と改良が行われています。本書で言及されている製品とその利用方法に関する記載事項について、ARM は保証しません。したがって当社では、製品の商品性または目的への適合性を含め、暗黙的・明示的に関係なく一切の保証を行いません。

1. 下記の条件に従い、ARM はこの ARM アーキテクチャ リファレンスマニュアルを次の目的に利用する永続的、非排他的、移転不可、無料、国際的なライセンスを許可します。使用目的は、(1) ARM からのライセンスにより配布されるマイクロプロセッサコアで実行することを目的としたソフトウェアアプリケーションとオペレーティングシステム、(2) ARM からのライセンスにより配布されるマイクロプロセッサコアで実行することを目的としたソフトウェアプログラムの開発用に設計されたツール、(3) ARM からのライセンスにより製造されるマイクロプロセッサコアを搭載する集積回路のいずれかの開発に限られます。

2. 条項 1 で明示的に与えられているものを除き、ARM アーキテクチャ リファレンスマニュアル、またはそれに含まれるいかなる知的著作物についても、いかなる権利、資格、利益も与えるものではありません。条項 1 に示されている許諾は、いかなる場合でも明示的、暗黙的、禁反言、その他の形で ARM アーキテクチャ リファレンスマニュアル以外のいかなる ARM テクノロジに関するライセンスも与えるものではありません。条項 1 で与えられるライセンスには、ARM パテントを使用する、または使用に含める権利は明示的に除外されます。条項 1 の条件では、次に示す権利は与えられません。(1) ARM アーキテクチャ リファレンスマニュアルを、この ARM アーキテクチャ リファレンスマニュアルで説明されている命令、プログラマ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

in whole or in part this ARM Architecture Reference Manual to third parties, other than to your subcontractors for the purposes of having developed products in accordance with the licence grant in Clause 1 without the express written permission of ARM; or (iv) translate or have translated this ARM Architecture Reference Manual into any other languages.

3. THE ARM ARCHITECTURE REFERENCE MANUAL IS PROVIDED "AS IS" WITH NO WARRANTIES EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF SATISFACTORY QUALITY, NONINFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE.

4. No licence, express, implied or otherwise, is granted to LICENSEE, under the provisions of Clause 1, to use the ARM tradename, in connection with the use of the ARM Architecture Reference Manual or any products based thereon. Nothing in Clause 1 shall be construed as authority for you to make any representations on behalf of ARM in respect of the ARM Architecture Reference Manual or any products based thereon.

Where the term ARM is used to refer to the company it means “ARM or any of its subsidiaries as appropriate”.

Note

The term ARM is also used to refer to versions of the ARM architecture, for example ARMv6 refers to version 6 of the ARM architecture. The context makes it clear when the term is used in this way.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited

110 Fulbourn Road Cambridge, England CB1 9NJ

Restricted Rights Legend: Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

This document is Non-Confidential. The right to use, copy and disclose this document is subject to the licence set out above.

抄訳

モデル、または両方に完全に、または部分的に互換であるマイクロプロセッサコアやモデルを開発する目的に使用すること。(2) ARM により、または ARM のために設計されたマイクロプロセッサコアのモデルを開発すること。(3) ARM からの文書による許諾なしに、この ARM アーキテクチャ リファレンスマニュアルの全体または一部をサードパーティに配布すること。ただし、条項 1 で許諾されたライセンスに従った製品を開発する目的で再契約者に提供する場合は除外されます。(4) この ARM アーキテクチャ リファレンスマニュアルを他の言語に翻訳すること。

3. ARM アーキテクチャ リファレンスマニュアルは現状のままで提供されるもので、明示的、暗黙的、法定にかかわらず一切の保証は行われません。これには、十分な品質、権利の不侵害、特定の目的への適合の保証を含みますが、それらに限定されるものではありません。

4. 条項 1 の条件に基づいて、明示的、暗黙的、その他にかかわらず、ARM の商標名を ARM アーキテクチャ リファレンスマニュアルまたはそれに基づいたいかなる製品とも組み合わせて使用する権利は与えられません。条項 1 のいかなる部分も、ARM アーキテクチャ リファレンスマニュアルや、それに基づいたいかなる製品についても、ARM を代表または代理するいかなる権限も与えるものではありません。

本書における ARM という用語は、「ARM、または該当する場合にはその子会社を含む」という意味で使用されています。

注

ARM という用語は、ARM アーキテクチャのバージョンを指すためにも使用されています。例えば、ARMv6 という用語は ARM アーキテクチャのバージョン 6 を表します。ARM という用語がこの意味で使用される場合、文脈により明白に示されています。

Copyright ©1996-1998, 2000, 2004-2008 ARM Limited

110 Fulbourn Road Cambridge, England CB1 9NJ

権利の制限事項：米国政府による使用、複製、公開は、DFARS 252.227-7013 (c) (1) (ii) および FAR 52.227-19 で規定されている制限の対象となります。

このドキュメントは公開文書です。本書を使用、コピー、および公開する権利は、上に記載されているライセンスの条件に従います。

Contents
ARM Architecture Reference Manual
ARMv7-A and ARMv7-R edition

Preface
About this manual xiv
Using this manual xv
Conventions xviii
Further reading xx
Feedback xxi

Part A **Application Level Architecture**

Chapter A1 **Introduction to the ARM Architecture**
A1.1 About the ARM architecture A1-2
A1.2 The ARM and Thumb instruction sets A1-3
A1.3 Architecture versions, profiles, and variants A1-4
A1.4 Architecture extensions A1-6
A1.5 The ARM memory model A1-7
A1.6 Debug A1-8

Chapter A2 **Application Level Programmers' Model**
A2.1 About the Application level programmers' model A2-2

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

目次
ARM アーキテクチャリファレンス マニュアル
ARMv7-A および ARMv7-R エディション

序章
本書について xiv
本書の使用法 xv
表記規則 xviii
参考資料 xx
ご意見・ご質問 xxi

パート A **アプリケーションレベルのアーキテクチャ**

A1 章 **ARM アーキテクチャの概要**
A1.1 ARM アーキテクチャについて A1-2
A1.2 ARM 命令セットと Thumb 命令セット A1-3
A1.3 アーキテクチャのバージョン、プロファイル、バリアント A1-4
A1.4 アーキテクチャ拡張機能 A1-6
A1.5 ARM メモリモデル A1-7
A1.6 デバッグ A1-8

A2 章 **アプリケーションレベルのプログラマモデル**
A2.1 アプリケーションレベルのプログラマモデルについて A2-2

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.2 ARM core data types and arithmetic A2-3
A2.3 ARM core registers A2-11
A2.4 The Application Program Status Register (APSR) A2-14
A2.5 Execution state registers A2-15
A2.6 Advanced SIMD and VFP extensions A2-20
A2.7 Floating-point data types and arithmetic A2-32
A2.8 Polynomial arithmetic over {0,1} A2-67
A2.9 Coprocessor support A2-68
A2.10 Execution environment support A2-69
A2.11 Exceptions, debug events and checks A2-81

Chapter A3 Application Level Memory Model
A3.1 Address space A3-2
A3.2 Alignment support A3-4
A3.3 Endian support A3-7
A3.4 Synchronization and semaphores A3-12
A3.5 Memory types and attributes and the memory order model A3-24
A3.6 Access rights A3-38
A3.7 Virtual and physical addressing A3-40
A3.8 Memory access order A3-41
A3.9 Caches and memory hierarchy A3-51

Chapter A4 The Instruction Sets
A4.1 About the instruction sets A4-2
A4.2 Unified Assembler Language A4-4
A4.3 Branch instructions A4-7
A4.4 Data-processing instructions A4-8
A4.5 Status register access instructions A4-18
A4.6 Load/store instructions A4-19
A4.7 Load/store multiple instructions A4-22
A4.8 Miscellaneous instructions A4-23
A4.9 Exception-generating and exception-handling instructions A4-24
A4.10 Coprocessor instructions A4-25
A4.11 Advanced SIMD and VFP load/store instructions A4-26
A4.12 Advanced SIMD and VFP register transfer instructions A4-29
A4.13 Advanced SIMD data-processing operations A4-30
A4.14 VFP data-processing instructions A4-38

Chapter A5 ARM Instruction Set Encoding
A5.1 ARM instruction set encoding A5-2
A5.2 Data-processing and miscellaneous instructions A5-4
A5.3 Load/store word and unsigned byte A5-19
A5.4 Media instructions A5-21
A5.5 Branch, branch with link, and block data transfer A5-27
A5.6 Supervisor Call, and coprocessor instructions A5-28
A5.7 Unconditional instructions A5-30

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A2.2 ARM コアのデータ型と演算 A2-3
A2.3 ARM コアレジスタ A2-11
A2.4 アプリケーションプログラム ステータスレジスタ (APSR) A2-14
A2.5 実行状態レジスタ A2-15
A2.6 アドバンスト SIMD および VFP 拡張機能 A2-20
A2.7 浮動小数点データ型と算術演算 A2-32
A2.8 {0,1} における多項式算術演算 A2-67
A2.9 コプロセッサのサポート A2-68
A2.10 実行環境のサポート A2-69
A2.11 例外、デバッグイベント、チェック A2-81

A3 章 アプリケーションレベルのメモリモデル
A3.1 アドレス空間 A3-2
A3.2 アライメントのサポート A3-4
A3.3 エンディアンのサポート A3-7
A3.4 同期化とセマフォ A3-12
A3.5 メモリのタイプおよび属性とメモリオーダ モデル A3-24
A3.6 アクセス権 A3-38
A3.7 仮想アドレシングと物理アドレシング A3-40
A3.8 メモリアクセスの順序 A3-41
A3.9 キャッシュとメモリ階層 A3-51

A4 章 命令セット
A4.1 命令セットについて A4-2
A4.2 統一アセンブラ言語 A4-4
A4.3 分岐命令 A4-7
A4.4 データ処理命令 A4-8
A4.5 ステータスレジスタ アクセス命令 A4-18
A4.6 ロード / ストア命令 A4-19
A4.7 複数ロード / ストア命令 A4-22
A4.8 その他の命令 A4-23
A4.9 例外生成命令と例外処理命令 A4-24
A4.10 コプロセッサ命令 A4-25
A4.11 アドバンスト SIMD および VFP のロード / ストア命令 A4-26
A4.12 アドバンスト SIMD および VFP のレジスタ転送命令 A4-29
A4.13 アドバンスト SIMD のデータ処理操作 A4-30
A4.14 VFP のデータ処理命令 A4-38

A5 章 ARM 命令セットのエンコード
A5.1 ARM 命令セットのエンコード A5-2
A5.2 データ処理命令とその他の命令 A5-4
A5.3 ワードおよび符号なしバイトのロード / ストア A5-19
A5.4 メディア命令 A5-21
A5.5 分岐、リンク付き分岐、ブロックデータ転送 A5-27
A5.6 スーパバイザコールおよびコプロセッサ命令 A5-28
A5.7 無条件命令 A5-30

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter A6 **Thumb Instruction Set Encoding**
A6.1 Thumb instruction set encoding A6-2
A6.2 16-bit Thumb instruction encoding A6-6
A6.3 32-bit Thumb instruction encoding A6-14

Chapter A7 **Advanced SIMD and VFP Instruction Encoding**
A7.1 Overview A7-2
A7.2 Advanced SIMD and VFP instruction syntax A7-3
A7.3 Register encoding A7-8
A7.4 Advanced SIMD data-processing instructions A7-10
A7.5 VFP data-processing instructions A7-24
A7.6 Extension register load/store instructions A7-26
A7.7 Advanced SIMD element or structure load/store instructions A7-27
A7.8 8, 16, and 32-bit transfer between ARM core and extension registers A7-31
A7.9 64-bit transfers between ARM core and extension registers A7-32

Chapter A8 **Instruction Details**
A8.1 Format of instruction descriptions A8-2
A8.2 Standard assembler syntax fields A8-7
A8.3 Conditional execution A8-8
A8.4 Shifts applied to a register A8-10
A8.5 Memory accesses A8-13
A8.6 Alphabetical list of instructions A8-14

Chapter A9 **ThumbEE**
A9.1 The ThumbEE instruction set A9-2
A9.2 ThumbEE instruction set encoding A9-6
A9.3 Additional instructions in Thumb and ThumbEE instruction sets A9-7
A9.4 ThumbEE instructions with modified behavior A9-8
A9.5 Additional ThumbEE instructions A9-14

Part B **System Level Architecture**

Chapter B1 **The System Level Programmers' Model**
B1.1 About the system level programmers' model B1-2
B1.2 System level concepts and terminology B1-3
B1.3 ARM processor modes and core registers B1-6
B1.4 Instruction set states B1-23
B1.5 The Security Extensions B1-25
B1.6 Exceptions B1-30
B1.7 Coprocessors and system control B1-62
B1.8 Advanced SIMD and floating-point support B1-64
B1.9 Execution environment support B1-73

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6 章 **Thumb 命令セットのエンコード**
A6.1 Thumb 命令セットのエンコード A6-2
A6.2 16 ビット Thumb 命令のエンコード A6-6
A6.3 32 ビット Thumb 命令のエンコード A6-14

A7 章 **アドバンスト SIMD および VFP 命令のエンコード**
A7.1 概要 A7-2
A7.2 アドバンスト SIMD および VFP 命令の構文 A7-3
A7.3 レジスタのエンコード A7-8
A7.4 アドバンスト SIMD のデータ処理命令 A7-10
A7.5 VFP のデータ処理命令 A7-24
A7.6 拡張機能レジスタロード / ストア命令 A7-26
A7.7 アドバンスト SIMD のエレメントまたは構造ロード / ストア命令 A7-27
A7.8 ARM コアレジスタと拡張機能レジスタとの 8、16、32 ビット転送 A7-31
A7.9 ARM コアレジスタと拡張機能レジスタとの 64 ビット転送 A7-32

A8 章 **命令の詳細**
A8.1 命令の説明フォーマット A8-2
A8.2 標準アセンブラ構文のフィールド A8-7
A8.3 条件付き実行 A8-8
A8.4 レジスタに適用されるシフト A8-10
A8.5 メモリアクセス A8-13
A8.6 命令のアルファベット順リスト A8-14

A9 章 **ThumbEE**
A9.1 ThumbEE 命令セット A9-2
A9.2 ThumbEE 命令セットのエンコード A9-6
A9.3 Thumb 命令セットおよび ThumbEE 命令セットの追加命令 A9-7
A9.4 動作が変更された ThumbEE 命令 A9-8
A9.5 追加の ThumbEE 命令 A9-14

パート B **システムレベルのアーキテクチャ**

B1 章 **システムレベルのプログラマモデル**
B1.1 システムレベルのプログラマモデルについて B1-2
B1.2 システムレベルの概念と用語 B1-3
B1.3 ARM プロセッサモードとコアレジスタ B1-6
B1.4 命令セット状態 B1-23
B1.5 セキュリティ拡張機能 B1-25
B1.6 例外 B1-30
B1.7 コプロセッサとシステム制御 B1-62
B1.8 アドバンスト SIMD と浮動小数点のサポート B1-64
B1.9 実行環境のサポート B1-73

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter B2 **Common Memory System Architecture Features**
B2.1 About the memory system architecture B2-2
B2.2 Caches B2-3
B2.3 Implementation defined memory system features B2-27
B2.4 Pseudocode details of general memory system operations B2-29

Chapter B3 **Virtual Memory System Architecture (VMSA)**
B3.1 About the VMSA B3-2
B3.2 Memory access sequence B3-4
B3.3 Translation tables B3-7
B3.4 Address mapping restrictions B3-23
B3.5 Secure and Non-secure address spaces B3-26
B3.6 Memory access control B3-28
B3.7 Memory region attributes B3-32
B3.8 VMSA memory aborts B3-40
B3.9 Fault Status and Fault Address registers in a VMSA implementation B3-48
B3.10 Translation Lookaside Buffers (TLBs) B3-54
B3.11 Virtual Address to Physical Address translation operations B3-63
B3.12 CP15 registers for a VMSA implementation B3-64
B3.13 Pseudocode details of VMSA memory system operations B3-156

Chapter B4 **Protected Memory System Architecture (PMSA)**
B4.1 About the PMSA B4-2
B4.2 Memory access control B4-9
B4.3 Memory region attributes B4-11
B4.4 PMSA memory aborts B4-13
B4.5 Fault Status and Fault Address registers in a PMSA implementation B4-18
B4.6 CP15 registers for a PMSA implementation B4-22
B4.7 Pseudocode details of PMSA memory system operations B4-79

Chapter B5 **The CPUID Identification Scheme**
B5.1 Introduction to the CPUID scheme B5-2
B5.2 The CPUID registers B5-4
B5.3 Advanced SIMD and VFP feature identification registers B5-34

Chapter B6 **System Instructions**
B6.1 Alphabetical list of instructions B6-2

Part C **Debug Architecture**

Chapter C1 **Introduction to the ARM Debug Architecture**
C1.1 Scope of part C of this manual C1-2
C1.2 About the ARM Debug architecture C1-3

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B2 章 **一般的なメモリシステム アーキテクチャの機能**
B2.1 メモリシステム アーキテクチャについて B2-2
B2.2 キャッシュ B2-3
B2.3 実装定義のメモリシステムの機能 B2-27
B2.4 一般的なメモリシステム操作の擬似コードの詳細 B2-29

B3 章 **仮想メモリシステム アーキテクチャ (VMSA)**
B3.1 VMSA について B3-2
B3.2 メモリアクセス シーケンス B3-4
B3.3 変換テーブル B3-7
B3.4 アドレスマッピングの制限 B3-23
B3.5 セキュアアドレス空間と非セキュアアドレス空間 B3-26
B3.6 メモリアクセス制御 B3-28
B3.7 メモリ領域の属性 B3-32
B3.8 VMSA メモリアボート B3-40
B3.9 VMSA 実装のフォールトステータス レジスタと フォールトアドレス レジスタ B3-48
B3.10 変換ルックアサイドバッファ（ＴＬＢ） B3-54
B3.11 仮想アドレスから物理アドレスへの変換操作 B3-63
B3.12 VMSA 実装の CP15 レジスタ B3-64
B3.13 VMSA メモリシステム操作の擬似コードの詳細 B3-156

B4 章 **保護メモリシステム アーキテクチャ (PMSA)**
B4.1 PMSA について B4-2
B4.2 メモリアクセスの制御 B4-9
B4.3 メモリ領域の属性 B4-11
B4.4 PMSA メモリアボート B4-13
B4.5 PMSA 実装のフォールトステータス レジスタと フォールトアドレス レジスタ B4-18
B4.6 PMSA 実装の CP15 レジスタ B4-22
B4.7 PMSA メモリシステム操作の擬似コードの詳細 B4-79

B5 章 **CPUID 識別方式**
B5.1 CPUID 方式の概要 B5-2
B5.2 CPUID レジスタ B5-4
B5.3 アドバンスト SIMD および VFP 機能識別レジスタ B5-34

B6 章 **システム命令**
B6.1 命令のアルファベット順リスト B6-2

パート C **デバッグアーキテクチャ**

C1 章 **ARM デバッグアーキテクチャの概要**
C1.1 本書のパート C の範囲 C1-2
C1.2 ARM デバッグアーキテクチャについて C1-3

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C1.3 Security Extensions and debug C1-8
C1.4 Register interfaces C1-9

Chapter C2 Invasive Debug Authentication
C2.1 About invasive debug authentication C2-2

Chapter C3 Debug Events
C3.1 About debug events C3-2
C3.2 Software debug events C3-5
C3.3 Halting debug events C3-38
C3.4 Generation of debug events C3-40
C3.5 Debug event prioritization C3-43

Chapter C4 Debug Exceptions
C4.1 About debug exceptions C4-2
C4.2 Effects of debug exceptions on CP15 registers and the DBGWFAR C4-4

Chapter C5 Debug State
C5.1 About Debug state C5-2
C5.2 Entering Debug state C5-3
C5.3 Behavior of the PC and CPSR in Debug state C5-7
C5.4 Executing instructions in Debug state C5-9
C5.5 Privilege in Debug state C5-13
C5.6 Behavior of non-invasive debug in Debug state C5-19
C5.7 Exceptions in Debug state C5-20
C5.8 Memory system behavior in Debug state C5-24
C5.9 Leaving Debug state C5-28

Chapter C6 Debug Register Interfaces
C6.1 About the debug register interfaces C6-2
C6.2 Reset and power-down support C6-4
C6.3 Debug register map C6-18
C6.4 Synchronization of debug register updates C6-24
C6.5 Access permissions C6-26
C6.6 The CP14 debug register interfaces C6-32
C6.7 The memory-mapped and recommended external debug interfaces C6-43

Chapter C7 Non-invasive Debug Authentication
C7.1 About non-invasive debug authentication C7-2
C7.2 v7 Debug non-invasive debug authentication C7-4
C7.3 Effects of non-invasive debug authentication C7-6
C7.4 ARMv6 non-invasive debug authentication C7-8

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C1.3 セキュリティ拡張機能とデバッグ C1-8
C1.4 レジスタインタフェース C1-9

C2 章 侵襲性デバッグの認証
C2.1 侵襲性デバッグの認証について C2-2

C3 章 デバッグイベント
C3.1 デバッグイベントについて C3-2
C3.2 ソフトウェアデバッグ イベント C3-5
C3.3 ホールトデバッグ イベント C3-38
C3.4 デバッグ イベントの生成 C3-40
C3.5 デバッグイベントの優先度付け C3-43

C4 章 デバッグ例外
C4.1 デバッグ例外について C4-2
C4.2 CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響 C4-4

C5 章 デバッグ状態
C5.1 デバッグ状態について C5-2
C5.2 デバッグ状態への遷移 C5-3
C5.3 デバッグ状態での PC および CPSR の動作 C5-7
C5.4 デバッグ状態での命令の実行 C5-9
C5.5 デバッグ状態での特権 C5-13
C5.6 デバッグ状態での非侵襲性デバッグの動作 C5-19
C5.7 デバッグ状態での例外 C5-20
C5.8 デバッグ状態でのメモリシステムの動作 C5-24
C5.9 デバッグ状態の終了 C5-28

C6 章 デバッグレジスタ インタフェース
C6.1 デバッグレジスタ インタフェースについて C6-2
C6.2 リセットと電力オフのサポート C6-4
C6.3 デバッグレジスタのマップ C6-18
C6.4 デバッグレジスタの更新の同期 C6-23
C6.5 アクセス許可 C6-25
C6.6 CP14 デバッグレジスタ インタフェース C6-32
C6.7 メモリマップされたインタフェースと推奨される外部デバッグインタフェース C6-43

C7 章 非侵襲性デバッグの認証
C7.1 非侵襲性デバッグの認証について C7-2
C7.2 v7 デバッグでの非侵襲性デバッグの認証 C7-4
C7.3 非侵襲性デバッグ認証の影響 C7-6
C7.4 ARMv6 での非侵襲性デバッグ認証 C7-8

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter C8 **Sample-based Profiling**
C8.1 Program Counter sampling C8-2

Chapter C9 **Performance Monitors**
C9.1 About the performance monitors C9-2
C9.2 Status in the ARM architecture C9-4
C9.3 Accuracy of the performance monitors C9-5
C9.4 Behavior on overflow C9-6
C9.5 Interaction with Security Extensions C9-7
C9.6 Interaction with trace C9-8
C9.7 Interaction with power saving operations C9-9
C9.8 CP15 c9 register map C9-10
C9.9 Access permissions C9-12
C9.10 Event numbers C9-13

Chapter C10 **Debug Registers Reference**
C10.1 Accessing the debug registers C10-2
C10.2 Debug identification registers C10-3
C10.3 Control and status registers C10-10
C10.4 Instruction and data transfer registers C10-40
C10.5 Software debug event registers C10-48
C10.6 OS Save and Restore registers, v7 Debug only C10-75
C10.7 Memory system control registers C10-80
C10.8 Management registers, ARMv7 only C10-88
C10.9 Performance monitor registers C10-105

Appendix A **Recommended External Debug Interface**
A.1 System integration signals AppxA-2
A.2 Recommended debug slave port AppxA-13

Appendix B **Common VFP Subarchitecture Specification**
B.1 Scope of this appendix AppxB-2
B.2 Introduction to the Common VFP subarchitecture AppxB-3
B.3 Exception processing AppxB-6
B.4 Support code requirements AppxB-11
B.5 Context switching AppxB-14
B.6 Subarchitecture additions to the VFP system registers AppxB-15
B.7 Version 1 of the Common VFP subarchitecture AppxB-23
B.8 Version 2 of the Common VFP subarchitecture AppxB-24

Appendix C **Legacy Instruction Mnemonics**
C.1 Thumb instruction mnemonics AppxC-2
C.2 Pre-UAL pseudo-instruction NOP AppxC-3

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C8 章 **サンプルベースのプロファイリング**
C8.1 プログラムカウンタのサンプリング C8-2

C9 章 **パフォーマンスモニタ**
C9.1 パフォーマンスモニタについて C9-2
C9.2 ARM アーキテクチャでのステータス C9-4
C9.3 パフォーマンスモニタの正確性 C9-5
C9.4 オーバフロー時の動作 C9-6
C9.5 セキュリティ拡張機能との相互動作 C9-7
C9.6 トレースとの相互動作 C9-8
C9.7 電力削減動作との相互動作 C9-9
C9.8 CP15 c9 レジスタマップ C9-10
C9.9 アクセス許可 C9-12
C9.10 イベント番号 C9-13

C10 章 **デバッグレジスタのインタフェース**
C10.1 デバッグレジスタへのアクセス C10-2
C10.2 デバッグ識別レジスタ C10-3
C10.3 制御およびステータスレジスタ C10-10
C10.4 命令およびデータ転送レジスタ C10-40
C10.5 ソフトウェアデバッグイベント レジスタ C10-48
C10.6 OS 保存および復元レジスタ（v7 デバッグのみ） C10-75
C10.7 メモリシステム制御レジスタ C10-80
C10.8 管理レジスタ（ARMv7 のみ） C10-88
C10.9 パフォーマンスモニタ レジスタ C10-105

付録 A **推奨される外部デバッグインタフェース**
A.1 システム統合信号 ..... 付録 A-2
A.2 推奨されるデバッグ スレーブポート ..... 付録 A-13

付録 B **一般的な VFP サブアーキテクチャ仕様**
B.1 この付録の範囲 ..... 付録 B-2
B.2 一般的な VFP サブアーキテクチャの概要 ..... 付録 B-3
B.3 例外処理 ..... 付録 B-6
B.4 サポートコードの要件 ..... 付録 B-11
B.5 コンテキストスイッチ ..... 付録 B-14
B.6 VFP システムレジスタへのサブアーキテクチャ固有の追加 ..... 付録 B-15
B.7 一般的な VFP サブアーキテクチャのバージョン 1 ..... 付録 B-23
B.8 一般的な VFP サブアーキテクチャのバージョン 2 ..... 付録 B-24

付録 C **過去の命令のニーモニック**
C.1 Thumb 命令ニーモニック ..... 付録 C-2
C.2 UAL 以前の擬似命令 NOP ..... 付録 C-3

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Appendix D Deprecated and Obsolete Features
D.1 Deprecated features AppxD-2
D.2 Deprecated terminology AppxD-5
D.3 Obsolete features AppxD-6
D.4 Semaphore instructions AppxD-7
D.5 Use of the SP as a general-purpose register AppxD-8
D.6 Explicit use of the PC in ARM instructions AppxD-9
D.7 Deprecated Thumb instructions AppxD-10

Appendix E Fast Context Switch Extension (FCSE)
E.1 About the FCSE AppxE-2
E.2 Modified virtual addresses AppxE-3
E.3 Debug and trace AppxE-5

Appendix F VFP Vector Operation Support
F.1 About VFP vector mode AppxF-2
F.2 Vector length and stride control AppxF-3
F.3 VFP register banks AppxF-5
F.4 VFP instruction type selection AppxF-7

Appendix G ARMv6 Differences
G.1 Introduction to ARMv6 ..... AppxG-2
G.2 Application level register support AppxG-3
G.3 Application level memory support AppxG-6
G.4 Instruction set support AppxG-10
G.5 System level register support AppxG-16
G.6 System level memory model AppxG-20
G.7 System Control coprocessor (CP15) support AppxG-29

Appendix H ARMv4 and ARMv5 Differences
H.1 Introduction to ARMv4 and ARMv5 ..... AppxH-2
H.2 Application level register support AppxH-4
H.3 Application level memory support AppxH-6
H.4 Instruction set support AppxH-11
H.5 System level register support AppxH-18
H.6 System level memory model AppxH-21
H.7 System Control coprocessor (CP15) support AppxH-31

Appendix I Pseudocode Definition
I.1 Instruction encoding diagrams and pseudocode AppxI-2
I.2 Limitations of pseudocode AppxI-4
I.3 Data types AppxI-5
I.4 Expressions AppxI-9
I.5 Operators and built-in functions AppxI-11
I.6 Statements and program structure AppxI-17
I.7 Miscellaneous helper procedures and functions AppxI-22

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

付録 D 非推奨または廃止された機能
D.1 非推奨の機能 ..... 付録 D-2
D.2 非推奨の用語 ..... 付録 D-5
D.3 廃止された機能 ..... 付録 D-6
D.4 セマフォ命令 ..... 付録 D-7
D.5 SP の汎用レジスタとしての使用 ..... 付録 D-8
D.6 ARM 命令での PC の明示的な使用 ..... 付録 D-9
D.7 非推奨の Thumb 命令 ..... 付録 D-10

付録 E 高速コンテキストスイッチ拡張機能 (FCSE)
E.1 FCSE について ..... 付録 E-2
E.2 修飾仮想アドレス ..... 付録 E-3
E.3 デバッグおよびトレース ..... 付録 E-5

付録 F VFP ベクタ演算のサポート
F.1 VFP ベクタモードについて ..... 付録 F-2
F.2 ベクタ長とストライドの制御 ..... 付録 F-3
F.3 VFP レジスタバンク ..... 付録 F-5
F.4 VFP 命令タイプの選択 ..... 付録 F-7

付録 G ARMv6 との相違点
G.1 ARMv6 の概要 ..... 付録 G-2
G.2 アプリケーションレベルのレジスタサポート ..... 付録 G-3
G.3 アプリケーションレベルのメモリサポート ..... 付録 G-6
G.4 命令セットのサポート ..... 付録 G-10
G.5 システムレベルのレジスタサポート ..... 付録 G-16
G.6 システムレベルのメモリモデル ..... 付録 G-20
G.7 システム制御コプロセッサ (CP15) のサポート ..... 付録 G-29

付録 H ARMv4 および ARMv5 での相違点
H.1 ARMv4 および ARMv5 の概要 ..... 付録 H-2
H.2 アプリケーションレベルのレジスタサポート ..... 付録 H-4
H.3 アプリケーションレベルのメモリサポート ..... 付録 H-6
H.4 命令セットのサポート ..... 付録 H-11
H.5 システムレベル レジスタのサポート ..... 付録 H-18
H.6 システムレベルのメモリモデル ..... 付録 H-21
H.7 システム制御コプロセッサ (CP15) のサポート ..... 付録 H-31

付録 I 擬似コードの定義
I.1 命令エンコード図および擬似コード ..... 付録 I-2
I.2 擬似コードの制限 ..... 付録 I-4
I.3 データ型 ..... 付録 I-5
I.4 式 ..... 付録 I-9
I.5 演算子と組込み関数 ..... 付録 I-11
I.6 文とプログラム構造 ..... 付録 I-17
I.7 その他のヘルパプロシージャおよび関数 ..... 付録 I-22

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Appendix J **Pseudocode Index**
J.1 Pseudocode operators and keywords AppxJ-2
J.2 Pseudocode functions and procedures AppxJ-6

Appendix K **Register Index**
K.1 Register index AppxK-2

Glossary

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

付録 J **擬似コード索引**
J.1 擬似コードの演算子とキーワード ..... 付録 J-2
J.2 擬似コードの関数とプロシージャ ..... 付録 J-6

付録 K **レジスタ索引**
K.1 レジスタ索引 ..... 付録 K-2

用語集

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Preface

This preface summarizes the contents of this manual and lists the conventions it uses. It contains the following sections:

- *About this manual* on page xiv
- *Using this manual* on page xv
- *Conventions* on page xviii
- *Further reading* on page xx
- *Feedback* on page xxi.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

序章

本章では、本書の内容について説明し、使用する表記規則のリストを示します。本章は次のセクションから構成されています。

- *本書について* P. xiv
- *本書の使用法* P. xv
- *表記規則* P. xviii
- *参考資料* P. xx
- *ご意見・ご質問* P. xxi

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

About this manual

This manual describes the ARM®v7 instruction set architecture, including its high code density Thumb® instruction encoding and the following extensions to it:

- The System Control coprocessor, coprocessor 15 (CP15), used to control memory system components such as caches, write buffers, Memory Management Units, and Protection Units.
- The optional Advanced SIMD extension, that provides high-performance integer and single-precision floating-point vector operations.
- The optional VFP extension, that provides high-performance floating-point operations. It can optionally support double-precision operations.
- The Debug architecture, that provides software access to debug features in ARM processors.

Part A describes the application level view of the architecture. It describes the application level view of the programmers' model and the memory model. It also describes the precise effects of each instruction in *User mode* (the normal operating mode), including any restrictions on its use. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate ARM machine code.

Part B describes the system level view of the architecture. It gives details of system registers that are not accessible from User mode, and the system level view of the memory model. It also gives full details of the effects of instructions in *privileged modes* (any mode other than User mode), where these are different from their effects in User mode.

Part C describes the Debug architecture. This is an extension to the ARM architecture that provides configuration, breakpoint and watchpoint support, and a *Debug Communications Channel* (DCC) to a debug host.

Assembler syntax is given for the instructions described in this manual, permitting instructions to be specified in textual form. However, this manual is not intended as tutorial material for ARM assembler language, nor does it describe ARM assembler language at anything other than a very basic level. To make effective use of ARM assembler language, consult the documentation supplied with the assembler being used.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

本書について

本書では、ARM®v7 命令セットアーキテクチャについて説明します。これには、コード密度の高い Thumb® 命令エンコードと、次の拡張機能も含まれています。

- システム制御コプロセッサ、コプロセッサ 15 (CP15)。キャッシュ、ライトバッファ、メモリ管理ユニット、保護ユニットなどのメモリシステムコンポーネントを制御するために使用されます。
- オプションのアドバンスト SIMD 拡張機能。高パフォーマンスの整数および単精度浮動小数点ベクタ演算機能を提供します。
- オプションの VFP 拡張機能。高パフォーマンスの浮動小数点演算機能を提供します。倍精度演算をサポートすることもできます。
- デバッグアーキテクチャ。ARM プロセッサのデバッグ機能へのソフトウェアアクセスを提供します。

パート A では、アーキテクチャのアプリケーションレベルのビューについて説明します。プログラマモデルとメモリモデルについて、アプリケーションレベルのビューを解説します。また、*ユーザモード*（通常の動作モード）での各命令の正確な効果と、使用に関する制限について説明します。この情報は、コンパイラ、アセンブラ、および ARM 機械コードを生成する他のプログラムの作成者とユーザを主な対象としています。

パート B では、アーキテクチャのシステムレベルのビューについて説明します。ユーザモードからはアクセスできないシステムレジスタの詳細と、メモリモデルのシステムレベルのビューについて解説します。また、*特権モード*（ユーザモード以外のすべてのモード）での命令の効果に関する完全な詳細と、ユーザモードとの効果の相違点について説明します。

パート C では、デバッグアーキテクチャについて説明します。これは ARM アーキテクチャの拡張機能で、構成、ブレークポイントとウォッチポイントのサポート、およびデバッグホストへの*デバッグ通信チャネル* (DCC) を提供します。

本書に記載されている命令にはアセンブラ構文が与えられ、命令をテキスト形式で指定できるようになっています。ただし、本書は ARM アセンブラ言語のチュートリアル資料を意図したものではなく、ARM アセンブラ言語について非常に基本的なレベルの解説しか行っていません。ARM アセンブラ言語を効果的に使用するには、お使いのアセンブラに付属しているドキュメントを参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Using this manual

The information in this manual is organized into four parts, as described below.

Part A, Application Level Architecture

Part A describes the application level view of the architecture. It contains the following chapters:

Chapter A1 Gives a brief overview of the ARM architecture, and the ARM and Thumb instruction sets.

Chapter A2 Describes the application level view of the ARM programmers' model, including the application level view of the Advanced SIMD and VFP extensions. It describes the types of value that ARM instructions operate on, the general-purpose registers that contain those values, and the Application Program Status Register.

Chapter A3 Describes the application level view of the memory model, including the ARM memory types and attributes, and memory access control.

Chapter A4 Describes the range of instructions available in the ARM, Thumb, Advanced SIMD, and VFP instruction sets. It also contains some details of instruction operation, where these are common to several instructions.

Chapter A5 Gives details of the encoding of the ARM instruction set.

Chapter A6 Gives details of the encoding of the Thumb instruction set.

Chapter A7 Gives details of the encoding of the Advanced SIMD and VFP instruction sets.

Chapter A8 Provides detailed reference information about every instruction available in the Thumb, ARM, Advanced SIMD, and VFP instruction sets, with the exception of information only relevant in privileged modes.

Chapter A9 Provides detailed reference information about the ThumbEE (Execution Environment) variant of the Thumb instruction set.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

本書の使用法

本書の情報は、次に示す 4 つのパートに編成されています。

パート A、アプリケーションレベルのアーキテクチャ

パート A では、アーキテクチャのアプリケーションレベルのビューについて説明します。このパートは次の章から構成されています。

第 A1 章 ARM アーキテクチャと、ARM および Thumb 命令セットについて簡単に説明します。

第 A2 章 ARM プログラマモデルのアプリケーションレベルのビューについて、アドバンスト SIMD や VFP 拡張機能のアプリケーションレベルのビューも含めて説明します。ARM 命令で扱う値の型、それらの値を格納する汎用レジスタ、およびアプリケーションプログラム ステータスレジスタについて解説します。

第 A3 章 メモリモデルのアプリケーションレベルのビューについて、ARM のメモリタイプと属性、およびメモリアクセスの制御も含めて説明します。

第 A4 章 ARM、Thumb、アドバンスト SIMD、VFP 命令セットで利用できる範囲の命令について説明します。いくつかの命令で共通の部分については、命令の動作の詳細についても一部解説します。

第 A5 章 ARM 命令セットのエンコードの詳細について説明します。

第 A6 章 Thumb 命令セットのエンコードの詳細について説明します。

第 A7 章 アドバンスト SIMD および VFP 命令セットのエンコードの詳細について説明します。

第 A8 章 Thumb、ARM、アドバンスト SIMD、VFP 命令セットで使用できるすべての命令に関する詳細なリファレンス情報を、特権モードにのみ関連する情報を除いて記載します。

第 A9 章 Thumb 命令セットの Thumb EE（実行環境）バリアントに関する詳細なリファレンス情報を記載します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Part B, System Level Architecture

Part B describes the system level view of the architecture. It contains the following chapters:

Chapter B1 Describes the system level view of the programmers' model.

Chapter B2 Describes the system level view of the memory model features that are common to all memory systems.

Chapter B3 Describes the system level view of the Virtual Memory System Architecture (VMSA) that is part of all ARMv7-A implementations. This chapter includes descriptions of all of the CP15 System Control Coprocessor registers in a VMSA implementation.

Chapter B4 Describes the system level view of the Protected Memory System Architecture (PMSA) that is part of all ARMv7-R implementations. This chapter includes descriptions of all of the CP15 System Control Coprocessor registers in a PMSA implementation.

Chapter B5 Describes the CPUID scheme.

Chapter B6 Provides detailed reference information about system instructions, and more information about instructions where they behave differently in privileged modes.

Part C, Debug Architecture

Part C describes the Debug architecture. It contains the following chapters:

Chapter C1 Gives a brief introduction to the Debug architecture.

Chapter C2 Describes the authentication of invasive debug.

Chapter C3 Describes the debug events.

Chapter C4 Describes the debug exceptions.

Chapter C5 Describes Debug state.

Chapter C6 Describes the permitted debug register interfaces.

Chapter C7 Describes the authentication of non-invasive debug.

Chapter C8 Describes sample-based profiling.

Chapter C9 Describes the ARM performance monitors.

Chapter C10 Describes the debug registers.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

パート B、システムレベルのアーキテクチャ

パート B では、アーキテクチャのシステムレベルのビューについて説明します。このパートは次の章から構成されています。

第 B1 章 プログラマモデルのシステムレベルのビューについて説明します。

第 B2 章 すべてのメモリシステムに共通のメモリモデル機能に関して、システムレベルのビューについて説明します。

第 B3 章 すべての ARMv7-A 実装の一部である、仮想メモリシステム アーキテクチャ (VMSA) のシステムレベルのビューについて説明します。本章には、VMSA 実装の CP15 システム制御コプロセッサレジスタすべての説明が含まれています。

第 B4 章 すべての ARMv7-R 実装の一部である、保護メモリシステム アーキテクチャ (PMSA) のシステムレベルのビューについて説明します。本章には、PMSA 実装の CP15 システム制御コプロセッサレジスタすべての説明が含まれています。

第 B5 章 CPUID の識別方式について説明します。

第 B6 章 システム命令に関する詳細なリファレンス情報と、特権モードで動作が異なる命令に関する追加情報を記載します。

パート C、デバッグアーキテクチャ

パート C では、デバッグアーキテクチャについて説明します。このパートは次の章から構成されています。

第 C1 章 デバッグアーキテクチャについて簡単に紹介します。

第 C2 章 侵襲性デバッグの認証について説明します。

第 C3 章 デバッグイベントについて説明します。

第 C4 章 デバッグ例外について説明します。

第 C5 章 デバッグ状態について説明します。

第 C6 章 許可されるデバッグレジスタ インタフェースについて説明します。

第 C7 章 非侵襲性デバッグの認証について説明します。

第 C8 章 サンプルベースのプロファイリングについて説明します。

第 C9 章 ARM パフォーマンスモニタについて説明します。

第 C10 章 デバッグレジスタについて説明します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Part D, Appendices

This manual contains the following appendices:

Appendix A Describes the recommended external Debug interfaces.

Note

This description is not part of the ARM architecture specification. It is included here only as supplementary information, for the convenience of developers and users who might require this information.

Appendix B The Common VFP subarchitecture specification.

Note

This specification is not part of the ARM architecture specification. This sub-architectural information is included here only as supplementary information, for the convenience of developers and users who might require this information.

Appendix C Describes the legacy mnemonics.

Appendix D Identifies the deprecated architectural features.

Appendix E Describes the *Fast Context Switch Extension* (FCSE). From ARMv6, the use of this feature is deprecated, and in ARMv7 the FCSE is optional.

Appendix F Describes the VFP vector operations. Use of these operations is deprecated in ARMv7.

Appendix G Describes the differences in the ARMv6 architecture.

Appendix H Describes the differences in the ARMv4 and ARMv5 architectures.

Appendix I The formal definition of the pseudocode.

Appendix J Index to definitions of pseudocode operators, keywords, functions, and procedures.

Appendix K Index to register descriptions in the manual.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

パート D、付録

本書には次の付録が含まれています。

付録 A 推奨される外部デバッグインタフェースについて説明します。

注

この説明は、ARM アーキテクチャ仕様の一部ではありません。この情報を必要とする可能性がある開発者やユーザの便宜をはかるための補完情報としてのみ提供されるものです。

付録 B 一般的な VFP サブアーキテクチャの仕様。

注

この仕様は、ARM アーキテクチャ仕様の一部ではありません。このサブアーキテクチャ情報を必要とする可能性がある開発者やユーザの便宜をはかるための補完情報としてのみ提供されるものです。

付録 C 過去のニーモニックについて説明します。

付録 D アーキテクチャの推奨されない機能を記載します。

付録 E *高速コンテキストスイッチ拡張機能* (FCSE) について説明します。この機能は ARMv6 から使用が非推奨になっており、ARMv7 では FCSE がオプションです。

付録 F VFP ベクタ演算について説明します。ARMv7 では、これらの演算の使用は推奨されません。

付録 G ARMv6 アーキテクチャとの相違点について説明します。

付録 H ARMv4 および ARMv5 アーキテクチャとの相違点について説明します。

付録 I 擬似コードの正式な定義。

付録 J 擬似コードの演算子、キーワード、関数、プロシージャの定義への索引。

付録 K 本書に含まれているレジスタの説明への索引。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax descriptions

This manual contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `typewriter` font, and use the conventions described in *Assembler syntax* on page A8-4.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

擬似コードの説明

本書では、指定された機能を正確に説明するため、擬似コード形式を使用します。この擬似コードは `typewriter` フォントで表記されます。説明については、付録 I *「擬似コードの定義」*を参照して下さい。

アセンブラ構文の説明

本書には、アセンブラ命令およびアセンブラ命令のコンポーネントについて、多数の構文の説明が含まれています。これらは `typewriter` フォントで表記され、*アセンブラ構文* P. A8-4 に説明されている表記規則を使用します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Conventions

This manual employs typographic and other conventions intended to improve its ease of use.

General typographic conventions

`typewriter` Is used for assembler syntax descriptions, pseudocode descriptions of instructions, and source code examples. In the cases of assembler syntax descriptions and pseudocode descriptions, see the additional conventions below.

The `typewriter` style is also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode descriptions of instructions and source code examples.

italic Highlights important notes, introduces special terminology, and denotes internal cross-references and citations.

bold Is used for emphasis in descriptive lists and elsewhere, where appropriate.

SMALL CAPITALS Are used for a few terms that have specific technical meanings. Their meanings can be found in the *Glossary*.

Signals

In general this specification does not define processor signals, but it does include some signal examples and recommendations. It uses the following signal conventions:

Signal level The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals
- LOW for active-LOW signals.

Lower-case n At the start or end of a signal name denotes an active-LOW signal.

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by 0b, and hexadecimal numbers by `0x` and written in a `typewriter` font.

Bit values

Values of bits and bitfields are normally given in binary, in single quotes. The quotes are normally omitted in encoding diagrams and tables.

Pseudocode descriptions

This manual uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a `typewriter` font, and is described in Appendix I *Pseudocode Definition*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表記規則

本書では、各種の情報を見やすくするため、書体やその他の表記規則が採用されています。

書体の一般的な規則

`typewriter` アセンブラ構文の表記、命令の擬似コードの表記、ソースコードの例に使用されています。アセンブラ構文と擬似コードの表記については、以下に記載する追加の表記規則を参照して下さい。

`typewriter` スタイルは、命令ニーモニックのメインテキストと、アセンブラ構文の表記、命令の擬似コードの表記、ソースコードの例に出現する他の項目への参照にも使用されます。

斜体 重要な注釈の強調、特定の用語の初出時、本書内での相互参照と引用に使用されます。

太字 必要に応じて、説明のリストやその他の場所で強調のため使用されます。

小さい大文字 技術的に特別な意味があるいくつかの用語に使用されます。意味については、*「用語集」*を参照して下さい。

信号

一般に、この仕様ではプロセッサの信号は定義されていませんが、信号に関するいくつかの例と推奨事項が含まれています。信号には、次の信号表記規則が使用されます。

信号レベル アサートされる信号レベルは、その信号がアクティブ HIGH かアクティブ LOW かに依存します。「アサートされた (asserted)」とは、次の状態を意味します。

- アクティブ HIGH の信号に対しては HIGH
- アクティブ LOW の信号に対しては LOW

小文字の n アクティブ LOW 信号の信号名の最初または最後に付加されます。

数値

数値は通常、10 進数で記述されます。2 進数の数値の前には 0b、16 進数の数値の前には `0x` が付加され、これらは `typewriter` フォントで記載されます。

ビット値

ビットおよびビットフィールドの値は一般に、単一引用符でくくって 2 進数で表記されます。エンコード図と表では、通常は引用符を省略します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Further reading

This section lists publications from both ARM and third parties that provide more information on the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets and addenda, and the ARM Frequently Asked Questions.

ARM publications

- *ARM Debug Interface v5 Architecture Specification* (ARM IHI 0031)
- *ARMv7-M Architecture Reference Manual* (ARM DDI 0403)
- *CoreSight Architecture Specification* (ARM IHI 0029)
- *ARM Architecture Reference Manual* (ARM DDI 0100I)

  Note

  — Issue I of the *ARM Architecture Reference Manual* (DDI 0100I) was issued in July 2005 and describes the first version of the ARMv6 architecture, and all previous architecture versions.

  — Addison-Wesley Professional publish *ARM Architecture Reference Manual, Second Edition* (December 27, 2000). The contents of this are identical to Issue E of the *ARM Architecture Reference Manual* (DDI 0100E). It describes ARMv5TE and earlier versions of the ARM architecture, and is superseded by DDI 0100I.

- *Embedded Trace Macrocell Architecture Specification* (ARM IHI 0014)
- *CoreSight Program Flow Trace Architecture Specification* (ARM IHI 0035).

External publications

The following books are referred to in this manual, or provide more information:

- IEEE Std 1596.5-1993, *IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors*, ISBN 1-55937-354-7
- IEEE Std 1149.1-2001, *IEEE Standard Test Access Port and Boundary Scan Architecture (JTAG)*
- ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*
- JEP106, *Standard Manufacturers Identification Code*, JEDEC Solid State Technology Association
- *The Java Virtual Machine Specification* Second Edition, Tim Lindholm and Frank Yellin, published by Addison Wesley (ISBN: 0-201-43294-3)
- *Memory Consistency Models for Shared Memory-Multiprocessors*, Kourosh Gharachorloo, Stanford University Technical Report CSL-TR-95-685

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

参考資料

このセクションでは、ARM ファミリのプロセッサに関する追加情報が記載されている ARM 社およびサードパーティの出版物を紹介します。

ARM 社は、自社の出版物の定期的な更新と修正を行っています。最新の正誤表、追補情報、ARM によく寄せられる質問 (FAQ) については、`http://www.arm.com` を参照して下さい。

ARM の刊行物

- *ARM デバッグインタフェース v5、アーキテクチャ仕様書* (ARM IHI 0031)
- *ARMv7-M アーキテクチャ リファレンスマニュアル* (ARM DDI 0403)
- *CoreSight アーキテクチャ仕様* (ARM IHI 0029)
- *ARM アーキテクチャ リファレンスマニュアル* (ARM DDI 0100I)

  注

  – *ARM アーキテクチャ リファレンスマニュアル* (DDI 0100I) のリリース I は 2005 年 7 月に発行され、ARMv6 アーキテクチャの最初のバージョン、および以前のすべてのバージョンのアーキテクチャについての説明が含まれています。

  – Addison-Wesley Professional 刊、*ARM Architecture Reference Manual, Second Edition* (2000 年 12 月 27 日)。この本の内容は、*ARM アーキテクチャ リファレンスマニュアル* (DDI 0100E) のリリース E と同一です。この本は、ARMv5TE およびそれ以前のバージョンの ARM アーキテクチャについて解説しているもので、DDI 0100I に置き換えられています。

- *ARM エンベデッド トレースマクロセル アーキテクチャ仕様書* (ARM IHI 0014)
- *CoreSight プログラムフロー トレースアーキテクチャ仕様* (ARM IHI 0035)

社外の出版物

以下の書籍は、本書で参照されている、またはより詳細な情報が記載されているものです。

- IEEE Std 1596.5-1993、*IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors*、ISBN 1-55937-354-7
- IEEE Std 1149.1-2001、*IEEE Standard Test Access Port and Boundary Scan Architecture (JTAG)*
- ANSI/IEEE Std 754-1985、*IEEE Standard for Binary Floating-Point Arithmetic*
- JEP106、*Standard Manufacturers Identification Code*、JEDEC Solid State Technology Association
- *The Java Virtual Machine Specification* 第 2 版、Tim Lindholm and Frank Yellin、Addison Wesley 刊 (ISBN: 0-201-43294-3)
- *Memory Consistency Models for Shared Memory-Multiprocessors*、Kourosh Gharachorloo、Stanford University 技術レポート CSL-TR-95-685

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Feedback

ARM welcomes feedback on its documentation.

Feedback on this manual

If you notice any errors or omissions in this manual, send e-mail to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ご意見・ご質問

ARM 社では、本書に関するご意見・ご質問をお待ちしております。

本書に関するご意見

本書に関する誤り、記載もれなどがございましたら、電子メールに次の情報をご記入のうえ、`errata@arm.com` までお寄せ下さい。

- 資料名
- 資料番号
- ご意見のあるページ番号
- 問題についての簡潔なご説明

補足または改善すべき点についてのご提案もお待ちしております。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Part A

Application Level Architecture

抄訳

パート A

アプリケーションレベルのアーキテクチャ

Chapter A1
Introduction to the ARM Architecture

This chapter introduces the ARM architecture and contains the following sections:

- *About the ARM architecture* on page A1-2
- *The ARM and Thumb instruction sets* on page A1-3
- *Architecture versions, profiles, and variants* on page A1-4
- *Architecture extensions* on page A1-6
- *The ARM memory model* on page A1-7
- *Debug* on page A1-8.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 A1 章
ARM アーキテクチャの概要

本章では、ARM アーキテクチャの概要について説明します。本章は次のセクションから構成されています。

- *ARM アーキテクチャについて* P. A1-2
- *ARM 命令セットと Thumb 命令セット* P. A1-3
- *アーキテクチャのバージョン、プロファイル、バリアント* P. A1-4
- *アーキテクチャ拡張機能* P. A1-6
- *ARM メモリモデル* P. A1-7
- *デバッグ* P. A1-8

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A1.1 About the ARM architecture

The ARM architecture supports implementations across a wide range of performance points. It is established as the dominant architecture in many market segments. The architectural simplicity of ARM processors leads to very small implementations, and small implementations mean devices can have very low power consumption. Implementation size, performance, and very low power consumption are key attributes of the ARM architecture.

The ARM architecture is a *Reduced Instruction Set Computer* (RISC) architecture, as it incorporates these typical RISC architecture features:

- a large uniform register file
- a *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents
- simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only.

In addition, the ARM architecture provides:

- instructions that combine a shift with an arithmetic or logical operation
- auto-increment and auto-decrement addressing modes to optimize program loops
- Load and Store Multiple instructions to maximize data throughput
- conditional execution of almost all instructions to maximize execution throughput.

These enhancements to a basic RISC architecture enable ARM processors to achieve a good balance of high performance, small code size, low power consumption, and small silicon area.

Except where the architecture specifies differently, the programmer-visible behavior of an implementation must be the same as a simple sequential execution of the program. This programmer-visible behavior does not include the execution time of the program.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A1.1 ARM アーキテクチャについて

ARM アーキテクチャは、幅広い性能レベルの実装に対応しています。多くの市場区分では、主流のアーキテクチャとして確立されています。ARM プロセッサのアーキテクチャは単純なため、非常に小型の実装が可能で、ひいてはデバイスの消費電力も非常に低く抑えることができます。実装サイズ、パフォーマンス、非常に低い消費電力が、ARM アーキテクチャの主な特長です。

ARM アーキテクチャは*縮小命令セットコンピュータ*(RISC) アーキテクチャで、次のような RISC アーキテクチャの標準的な特徴を備えています。

- 大型で同じ形式のレジスタファイル
- *ロード / ストア*アーキテクチャ。これは、データ処理操作が直接メモリの内容に対して行われるのではなく、レジスタの内容に対してのみ行われるアーキテクチャです。
- 単純なアドレシングモード。すべてのロード / ストアアドレスは、レジスタの内容と命令フィールドのみによって決定されます。

また、ARM アーキテクチャには次のような機能が含まれています。

- シフトと算術演算または論理演算を組み合わせる命令
- 自動インクリメントと自動デクリメントのアドレシングモードにより、プログラムループを最適化できます。
- 複数ロード / ストア命令により、データスループットを最大化できます。
- ほとんどの命令は条件付き実行が可能で、実行スループットを最大化できます。

基本的な RISC アーキテクチャにこのような拡張を加えることで、ARM プロセッサは高性能、小さなコードサイズ、低消費電力、小さなシリコン実装面積の適切なバランスを実現しています。

アーキテクチャが特に指定していない限り、プログラマから見える実装の動作は、プログラムの単純なシーケンシャル実行と同じでなければなりません。このプログラマから見える動作に、プログラムの実行時間は含まれません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A1.2 The ARM and Thumb instruction sets

The ARM instruction set is a set of 32-bit instructions providing comprehensive data-processing and control functions.

The Thumb instruction set was developed as a 16-bit instruction set with a subset of the functionality of the ARM instruction set. It provides significantly improved code density, at a cost of some reduction in performance. A processor executing Thumb instructions can change to executing ARM instructions for performance critical segments, in particular for handling interrupts.

In ARMv6T2, Thumb-2 technology is introduced. This technology makes it possible to extend the original Thumb instruction set with many 32-bit instructions. The range of 32-bit Thumb instructions included in ARMv6T2 permits Thumb code to achieve performance similar to ARM code, with code density better than that of earlier Thumb code.

From ARMv6T2, the ARM and Thumb instruction sets provide almost identical functionality. For more information, see Chapter A4 *The Instruction Sets*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A1.2 ARM 命令セットと Thumb 命令セット

ARM 命令セットは、包括的なデータ処理機能と制御機能を提供する 32 ビット命令セットです。

Thumb 命令セットは、ARM 命令セットの機能の一部を利用できる 16 ビット命令セットとして開発されました。Thumb 命令セットは、コード密度が非常に高い反面、パフォーマンスが多少劣ります。割り込み処理などパフォーマンスが重要視される部分については、プロセッサを Thumb 命令の実行から ARM 命令の実行に切り替えることができます。

ARMv6T2 では、Thumb-2 テクノロジが導入されました。このテクノロジにより、元の Thumb 命令セットに多くの 32 ビット命令を加えることが可能となりました。ARMv6T2 に含まれる 32 ビット Thumb 命令により、Thumb コードも ARM コードと同じパフォーマンスを持ちます。しかもコード密度は、従来の Thumb コードを上回ります。

ARMv6T2 およびそれ以後の ARM 命令セットと Thumb 命令セットは、ほぼ同じ機能を持っています。詳細については、第 A4 章*「命令セット」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A1.3 Architecture versions, profiles, and variants

The ARM and Thumb instruction set architectures have evolved significantly since they were first developed. They will continue to be developed in the future. Seven major versions of the instruction set have been defined to date, denoted by the version numbers 1 to 7. Of these, the first three versions are now obsolete.

ARMv7 provides three profiles:

ARMv7-A Application profile, described in this manual. Implements a traditional ARM architecture with multiple modes and supporting a *Virtual Memory System Architecture* (VMSA) based on an MMU. Supports the ARM and Thumb instruction sets.

ARMv7-R Real-time profile, described in this manual. Implements a traditional ARM architecture with multiple modes and supporting a *Protected Memory System Architecture* (PMSA) based on an MPU. Supports the ARM and Thumb instruction sets.

ARMv7-M Microcontroller profile, described in the ARMv7-M Architecture Reference Manual. Implements a programmers' model designed for fast interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages. Implements a variant of the ARMv7 PMSA and supports a variant of the Thumb instruction set.

Versions can be qualified with variant letters to specify additional instructions and other functionality that are included as an architecture extension. Extensions are typically included in the base architecture of the next version number. Provision is also made to exclude variants by prefixing the variant letter with x.

Some extensions are described separately instead of using a variant letter. For details of these extensions see *Architecture extensions* on page A1-6.

The valid variants of ARMv4, ARMv5, and ARMv6 are as follows:

ARMv4 The earliest architecture variant covered by this manual. It includes only the ARM instruction set.

ARMv4T Adds the Thumb instruction set.

ARMv5T Improves interworking of ARM and Thumb instructions. Adds count leading zeros (`CLZ`) and software breakpoint (`BKPT`) instructions.

ARMv5TE Enhances arithmetic support for digital signal processing (DSP) algorithms. Adds preload data (`PLD`), dual word load (`LDRD`), store (`STRD`), and 64-bit coprocessor register transfers (`MCRR`, `MRRC`).

ARMv5TEJ Adds the `BXJ` instruction and other support for the Jazelle® architecture extension.

ARMv6 Adds many new instructions to the ARM instruction set. Formalizes and revises the memory model and the Debug architecture.

ARMv6K Adds instructions to support multi-processing to the ARM instruction set, and some extra memory model features.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A1.3 アーキテクチャのバージョン、プロファイル、バリアント

ARM/Thumb 命令セットアーキテクチャは、最初に開発されて以来、大幅に進化しました。今後も開発は継続されます。現在までに命令セットは 7 つのメジャーバージョンが定義され、1 ～ 7 の番号が割り当てられています。このうち最初の 3 つは、現在使用されていません。

ARMv7 には 3 つのプロファイルがあります。

ARMv7-A 本書で解説するアプリケーションプロファイル。複数のモードを持つ従来の ARM アーキテクチャを実装し、MMU をベースとする*仮想メモリシステムアーキテクチャ*(VMSA) をサポートしています。ARM 命令セットと Thumb 命令セットをサポートしています。

ARMv7-R 本書で解説するリアルタイムプロファイル。複数のモードを持つ従来の ARM アーキテクチャを実装し、MPU をベースとする*保護メモリシステムアーキテクチャ*(PMSA) をサポートしています。ARM 命令セットと Thumb 命令セットをサポートしています。

ARMv7-M ARMv7-M アーキテクチャ リファレンスマニュアルで解説されているマイクロコントローラプロファイル。割り込みの高速処理を目的としたプログラマモデルを実装し、レジスタのハードウェアスタック機能を搭載し、高級言語による割り込みハンドラの記述に対応しています。ARMv7 PMSA のバリアントを実装し、Thumb 命令セットのバリアントをサポートしています。

各バージョンには、追加された命令やアーキテクチャ拡張に含まれる機能を示すバリアント文字が付加されていることがあります。拡張機能は通常、次のバージョン番号のベースアーキテクチャに搭載されます。バリアント文字の前に接頭文字 x が付いている場合は、そのバリアント以外を意味します。

一部の拡張機能は、バリアント文字を使うのではなく、独立して説明されています。詳細については、*アーキテクチャ拡張機能* P. A1-6 を参照して下さい。

ARMv4、ARMv5、ARMv6 の有効なバリアントは次のとおりです。

ARMv4 本書に出てくる中で最も古いアーキテクチャバリアント。ARM 命令セットだけを含みます。

ARMv4T Thumb 命令セットを追加しています。

ARMv5T ARM 命令と Thumb 命令のインターワーキングを改良しています。先行ゼロカウント (`CLZ`) 命令とソフトウェアブレークポイント (`BKPT`) 命令を追加しています。

ARMv5TE デジタル信号処理 (DSP) アルゴリズムに対する算術演算サポートを強化しています。データプリロード (`PLD`) 命令、デュアルワードのロード (`LDRD`) およびストア (`STRD`) 命令、64 ビットコプロセッサレジスタ転送 (`MCRR`、`MRRC`) 命令を追加しています。

ARMv5TEJ `BXJ` 命令をはじめ、Jazelle® アーキテクチャ拡張機能のサポートを追加しています。

ARMv6 ARM 命令セットに多くの新しい命令を追加しています。メモリモデルとデバッグアーキテクチャを正式に定め、改訂しています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARMv6T2 Introduces Thumb-2 technology, giving a major development of the Thumb instruction set to provide a similar level of functionality to the ARM instruction set.

Note

ARMv6KZ or ARMv6Z are sometimes used to describe the ARMv6K architecture with the optional Security Extensions.

For detailed information about versions of the ARM architecture, see Appendix G *ARMv6 Differences* and Appendix H *ARMv4 and ARMv5 Differences*.

The following architecture variants are now obsolete:

ARMv1, ARMv2, ARMv2a, ARMv3, ARMv3G, ARMv3M, ARMv4xM, ARMv4TxM, ARMv5, ARMv5xM, ARMv5TxM, and ARMv5TExP.

Contact ARM if you require details of obsolete variants.

Instruction descriptions in this manual specify the architecture versions that support them.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARMv6K マルチプロセッシングに対応する命令を ARM 命令セットに追加しています。メモリモデル機能もいくつか追加しています。

ARMv6T2 Thumb-2 テクノロジを導入しています。Thumb 命令セットが大きく拡張され、ARM 命令セットとほぼ同等の機能を提供するようになりました。

注

オプションのセキュリティ拡張機能を加えた ARMv6K アーキテクチャは、ARMv6KZ または ARMv6Z と呼ばれることがあります。

ARM アーキテクチャの各バージョンの詳細については、付録 G *「ARMv6 との相違点」*と付録 H *「ARMv4 および ARMv5 での相違点」*を参照して下さい。

次のアーキテクチャバリアントは、現在使用されていません。

ARMv1、ARMv2、ARMv2a、ARMv3、ARMv3G、ARMv3M、ARMv4xM、ARMv4TxM、ARMv5、ARMv5xM、ARMv5TxM、ARMv5TExP

廃止されたバリアントの詳細については、ARM に問い合わせ下さい。

本書では、各命令を説明する際に、その命令をサポートするアーキテクチャバージョンを記載しています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A1.4 Architecture extensions

This manual describes the following extensions to the ARM and Thumb instruction set architectures:

ThumbEE Is a variant of the Thumb instruction set that is designed as a target for dynamically generated code. It is:

- a required extension to the ARMv7-A profile
- an optional extension to the ARMv7-R profile.

VFP Is a floating-point coprocessor extension to the instruction set architectures. There have been three main versions of VFP to date:

- VFPv1 is obsolete. Details are available on request from ARM.
- VFPv2 is an optional extension to:
  - — the ARM instruction set in the ARMv5TE, ARMv5TEJ, ARMv6, and ARMv6K architectures
  - — the ARM and Thumb instruction sets in the ARMv6T2 architecture.
- VFPv3 is an optional extension to the ARM, Thumb and ThumbEE instruction sets in the ARMv7-A and ARMv7-R profiles.

  VFPv3 can be implemented with either thirty-two or sixteen doubleword registers, as described in *Advanced SIMD and VFP extension registers* on page A2-21. Where necessary, the terms VFPv3-D32 and VFPv3-D16 are used to distinguish between these two implementation options. Where the term VFPv3 is used it covers both options.

  VFPv3 can be extended by the half-precision extensions that provide conversion functions in both directions between half-precision floating-point and single-precision floating-point.

Advanced SIMD Is an instruction set extension that provides *Single Instruction Multiple Data* (SIMD) functionality. It is an optional extension to the ARMv7-A and ARMv7-R profiles. When VFPv3 and Advanced SIMD are both implemented, they use a shared register bank and have some shared instructions.

Advanced SIMD can be extended by the half-precision extensions that provide conversion functions in both directions between half-precision floating-point and single-precision floating-point.

Security Extensions Are a set of security features that facilitate the development of secure applications. They are an optional extension to the ARMv6K architecture and the ARMv7-A profile.

Jazelle Is the Java bytecode execution extension that extended ARMv5TE to ARMv5TEJ. From ARMv6 Jazelle is a required part of the architecture, but is still often described as the Jazelle extension.

Multiprocessing Extensions

Are a set of features that enhance multiprocessing functionality. They are an optional extension to the ARMv7-A and ARMv7-R profiles.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A1.4 アーキテクチャ拡張機能

本書では、ARM/Thumb 命令セットアーキテクチャの以下の拡張機能について説明します。

ThumbEE 動的に生成されるコードのターゲットとして設計された Thumb 命令セットのバリアントです。これは、

- ARMv7-A プロファイルでは必須の拡張機能です。
- ARMv7-R プロファイルではオプションの拡張機能です。

VFP 命令セットアーキテクチャの浮動小数点コプロセッサ拡張機能です。現在までに、VFP には 3 つの主要なバージョンがあります。

- VFPv1 は廃止されています。詳細については、ＡＲＭ に問い合わせ下さい。
- VFPv2 は、次の命令セットにおけるオプションの拡張機能です。
  - ARMv5TE、ARMv5TEJ、ARMv6、ARMv6K アーキテクチャの ARM 命令セット
  - ARMv6T2 アーキテクチャの ARM および Thumb 命令セット
- VFPv3 は、ARMv7-A および ARMv7-R プロファイルにおける ARM、Thumb、ThumbEE 命令セットのオプションの拡張機能です。

  VFPv3 は、*アドバンスト SIMD と VFP の拡張機能レジスタ* P. A2-21 で説明されているように、32 個または 16 個のダブルワードレジスタとして実装できます。2 つの実装オプションを区別する必要がある場合、それぞれ VFPv3-D32、VFPv3-D16 という用語を使用します。VFPv3 という用語を使用する場合は、両方のオプションを指します。

  VFPv3 には、半精度浮動小数点と単精度浮動小数点とを相互に変換する関数を含む、半精度の拡張機能を加えることができます。

アドバンスト SIMD *単一命令複数データ*(SIMD) 処理機能を持つ命令セット拡張機能です。これは、ARMv7-A および ARMv7-R プロファイルのオプションの拡張機能です。VFPv3 とアドバンスト SIMD の両方が実装されている場合、共有のレジスタバンクが使用され、一部の命令も共有されます。

アドバンスト SIMD には、半精度浮動小数点と単精度浮動小数点とを相互に変換する関数を含む、半精度の拡張機能を加えることができます。

セキュリティ拡張機能 セキュアアプリケーションの開発を促進するセキュリティ機能群です。これは、ARMv6K アーキテクチャおよび ARMv7-A プロファイルのオプションの拡張機能です。

Jazelle ARMv5TE から ARMv5TEJ への拡張点である、Java バイトコード実行拡張機能です。ARMv6 以降、Jazelle はアーキテクチャの必須部分ですが、依然として多くの場合に Jazelle 拡張と呼ばれます。

マルチプロセッシング拡張機能

マルチプロセッシング機能を強化する機能群です。これは、ARMv7-A および ARMv7-R プロファイルのオプションの拡張機能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A1.5 The ARM memory model

The ARM architecture uses a single, flat address space of 2^{32} 8-bit bytes. The address space is also regarded as 2^{30} 32-bit words or 2^{31} 16-bit halfwords.

The architecture provides facilities for:

- faulting unaligned memory accesses
- restricting access by applications to specified areas of memory
- translating virtual addresses provided by executing instructions into physical addresses
- altering the interpretation of word and halfword data between big-endian and little-endian
- optionally preventing out-of-order access to memory
- controlling caches
- synchronizing access to shared memory by multiple processors.

For more information, see:

- Chapter A3 *Application Level Memory Model*
- Chapter B2 *Common Memory System Architecture Features*
- Chapter B3 *Virtual Memory System Architecture (VMSA)*
- Chapter B4 *Protected Memory System Architecture (PMSA)*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A1.5 ARM メモリモデル

ARM アーキテクチャでは、2^{32} 個の 8 ビットバイトで構成される、単一のフラットなアドレス空間が使用されます。このアドレス空間は、2^{30} 個の 32 ビットワード、または 2^{31} 個の 16 ビットハーフワードと考えることもできます。

このアーキテクチャには、次のような機能があります。

- アンアライインド メモリアクセスでフォールトを生成する。
- メモリの特定の領域に対して、アプリケーションによるアクセスを制限する。
- 実行中の命令が提供する仮想アドレスを物理アドレスに変換する。
- ワードデータとハーフワードデータの解釈を、ビッグエンディアンとリトルエンディアンのいずれかに切り替える。
- （オプション） メモリへのアウトオブオーダー アクセスを防止する。
- キャッシュを制御する。
- 複数のプロセッサによる共有メモリへのアクセスを同期化する。

詳細については、次の各章を参照して下さい。

- 第 A3 章 *「アプリケーションレベルのメモリモデル」*
- 第 B2 章 *「一般的なメモリシステム アーキテクチャの機能」*
- 第 B3 章 *「仮想メモリシステム アーキテクチャ (VMSA)」*
- 第 B4 章 *「保護メモリシステム アーキテクチャ (PMSA)」*

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A1.6 Debug

ARMv7 processors implement two types of debug support:

Invasive debug Debug permitting modification of the state of the processor. This is intended primarily for run-control debugging.

Non-invasive debug Debug permitting data and program flow observation, without modifying the state of the processor or interrupting the flow of execution.

This provides for:

- instruction and data tracing
- program counter sampling
- performance monitors.

For more information, see Chapter C1 *Introduction to the ARM Debug Architecture*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A1.6 デバッグ

ARMv7 プロセッサには、次の 2 種類のデバッグサポートが実装されます。

侵襲性デバッグ プロセッサ状態の変更を許可するデバッグ。これは主に実行制御デバッグに使用します。

非侵襲性デバッグ プロセッサ状態を変更したり、実行フローに割り込んだりすることなく、データとプログラムのフローを観察するためのデバッグ。

このデバッグは、次のような目的に役立ちます。

- 命令とデータのトレース
- プログラムカウンタのサンプリング
- パフォーマンスモニタ

詳細については、第 C1 章「*ARM デバッグアーキテクチャの概要*」を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter A2
Application Level Programmers' Model

This chapter gives an application level view of the ARM programmers' model. It contains the following sections:

- *About the Application level programmers' model* on page A2-2
- *ARM core data types and arithmetic* on page A2-3
- *ARM core registers* on page A2-11
- *The Application Program Status Register (APSR)* on page A2-14
- *Execution state registers* on page A2-15
- *Advanced SIMD and VFP extensions* on page A2-20
- *Floating-point data types and arithmetic* on page A2-32
- *Polynomial arithmetic over {0,1}* on page A2-67
- *Coprocessor support* on page A2-68
- *Execution environment support* on page A2-69
- *Exceptions, debug events and checks* on page A2-81.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 A2 章
アプリケーションレベルのプログラマモデル

本章では、ARM プログラマモデルのアプリケーションレベルのビューについて説明します。本章は次のセクションから構成されています。

- *アプリケーションレベルのプログラマモデルについて* P. A2-2
- *ARM コアのデータ型と演算* P. A2-3
- *ARM コアレジスタ* P. A2-11
- *アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14
- *実行状態レジスタ* P. A2-15
- *アドバンスト SIMD および VFP 拡張機能* P. A2-20
- *浮動小数点データ型と算術演算* P. A2-32
- *{0,1} における多項式算術演算* P. A2-67
- *コプロセッサのサポート* P. A2-68
- *実行環境のサポート* P. A2-69
- *例外、デバッグイベント、チェック* P. A2-81

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.1 About the Application level programmers' model

This chapter contains the programmers' model information required for application development.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system. However, some knowledge of that system information is needed to put the Application level programmers' model into context.

System level support requires access to all features and facilities of the architecture, a mode of operation referred to as privileged operation. System code determines whether an application runs in a privileged or unprivileged manner. When an operating system supports both privileged and unprivileged operation, an application usually runs unprivileged. This:

- permits the operating system to allocate system resources to it in a unique or shared manner
- provides a degree of protection from other processes and tasks, and so helps protect the operating system from malfunctioning applications.

This chapter indicates where some system level understanding is helpful, and where appropriate it:

- gives an overview of the system level information
- gives references to the system level descriptions in Chapter B1 *The System Level Programmers' Model* and elsewhere.

The Security Extensions extend the architecture to provide hardware security features that support the development of secure applications. For more information, see *The Security Extensions* on page B1-25.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A2.1 アプリケーションレベルのプログラマモデルについて

本章には、アプリケーション開発に必要なプログラマモデル情報が含まれています。

本章の情報は、オペレーティングシステムのもとでのサービスおよびサポートアプリケーションの実行に必要なシステム情報とは明確に区別されています。しかし、アプリケーションレベルのプログラマモデルを理解するには、システム情報に関する多少の知識も必要です。

システムレベルのサポートでは、アーキテクチャのすべての機能や機構にアクセスすることが必要です。この動作モードを特権動作と呼びます。アプリケーションが特権または非特権のいずれの状態で実行されているかは、システムコードによって決定されます。オペレーティングシステムが特権と非特権の動作の両方をサポートしている場合、アプリケーションは一般に非特権で実行されます。これにより、次のような操作が可能になります。

- オペレーティングシステムは、システムリソースをアプリケーションに対して専用、または共用として割り当てることができます。
- 他のプロセスやタスクに対する一定の保護が得られ、オペレーティングシステムが動作不良のアプリケーションから保護されます。

本章では、ある程度のシステムレベルの理解が求められる場所ではその旨を記載します。また、必要に応じて次のような情報を記載します。

- システムレベル情報の概要
- 第 B1 章「*システムレベルのプログラマモデル*」および他の場所に記載されているシステムレベルの説明への参照

セキュリティ拡張機能は、アーキテクチャを拡張し、セキュアなアプリケーションの開発をサポートするハードウェアセキュリティ機能を提供するものです。詳細については、*セキュリティ拡張機能* P. B1-25 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.2 ARM core data types and arithmetic

All ARMv7-A and ARMv7-R processors support the following data types in memory:

Byte 8 bits
Halfword 16 bits
Word 32 bits
Doubleword 64 bits.

Processor registers are 32 bits in size. The instruction set contains instructions supporting the following data types held in registers:

- 32-bit pointers
- unsigned or signed 32-bit integers
- unsigned 16-bit or 8-bit integers, held in zero-extended form
- signed 16-bit or 8-bit integers, held in sign-extended form
- two 16-bit integers packed into a register
- four 8-bit integers packed into a register
- unsigned or signed 64-bit integers held in two registers.

Load and store operations can transfer bytes, halfwords, or words to and from memory. Loads of bytes or halfwords zero-extend or sign-extend the data as it is loaded, as specified in the appropriate load instruction.

The instruction sets include load and store operations that transfer two or more words to and from memory. You can load and store doublewords using these instructions. The exclusive doubleword load/store instructions `LDREXD` and `STREXD` specify single-copy atomic doubleword accesses to memory.

When any of the data types is described as *unsigned*, the N-bit data value represents a non-negative integer in the range 0 to 2^N-1, using normal binary format.

When any of these types is described as *signed*, the N-bit data value represents an integer in the range -2^{N-1} to $+2^{N-1}$-1, using two's complement format.

The instructions that operate on packed halfwords or bytes include some multiply instructions that use just one of two halfwords, and *Single Instruction Multiple Data* (SIMD) instructions that operate on all of the halfwords or bytes in parallel.

Direct instruction support for 64-bit integers is limited, and most 64-bit operations require sequences of two or more instructions to synthesize them.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.2 ARM コアのデータ型と演算

すべての ARMv7-A および ARMv7-R プロセッサは、メモリ上で次のデータ型をサポートしています。

バイト 8 ビット
ハーフワード 16 ビット
ワード 32 ビット
ダブルワード 64 ビット

プロセッサレジスタのサイズは 32 ビットです。命令セットには、レジスタに保持されている次のデータ型をサポートする命令が含まれています。

- 32 ビットポインタ
- 符号なしまたは符号付き 32 ビット整数
- 符号なし 16 ビットまたは 8 ビット整数、ゼロ拡張形式で保持
- 符号付き 16 ビットまたは 8 ビット整数、符号拡張形式で保持
- 1 つのレジスタにパックされた 2 つの 16 ビット整数
- 1 つのレジスタにパックされた 4 つの 8 ビット整数
- 符号なしまたは符号付き 64 ビット整数、2 つのレジスタに保持

ロード / ストア操作では、メモリとの間でバイト、ハーフワード、ワードの転送を実行できます。バイトやハーフワードのデータをロードする場合は、該当するロード命令の指定に従ってゼロ拡張または符号拡張が行われます。

命令セットには、2 つ以上のワードをメモリとの間でロード / ストアする操作が含まれています。これらの命令を使用して、ダブルワードをロード / ストアできます。ダブルワード排他ロード / ストア命令である `LDREXD` と `STREXD` は、メモリに対するシングルコピー アトミックのダブルワードアクセスを指定します。

*符号なし*と記載されている N ビットのデータ型は、通常のバイナリ形式を使用して、0 ～ $2^N - 1$ の範囲内にある負でない整数を表します。

*符号付き*と記載されている N ビットのデータ型は、2 の補数形式を使用して、-2^{N-1} ～ $+2^{N-1} - 1$ の範囲にある整数を表します。

パック化されたハーフワードまたはバイトに作用する命令には、1 つか 2 つのハーフワードしか使用しない乗算命令がいくつかと、すべてのハーフワードまたはバイトに並列的に演算を行う *単一命令複数データ*(SIMD) 命令が含まれます。

64 ビット整数を直接サポートする命令は限られており、大部分の 64 ビット演算は 2 つ以上の命令シーケンスを使用して合成する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.2.1 Integer arithmetic

The instruction set provides a wide variety of operations on the values in registers, including bitwise logical operations, shifts, additions, subtractions, multiplications, and many others. These operations are defined using the *pseudocode* described in Appendix I *Pseudocode Definition*, usually in one of three ways:

- By direct use of the pseudocode operators and built-in functions defined in *Operators and built-in functions* on page AppxI-11.
- By use of pseudocode helper functions defined in the main text. These can be located using the table in Appendix J *Pseudocode Index*.
- By a sequence of the form:
  1. Use of the `SInt()`, `UInt()`, and `Int()` built-in functions defined in *Converting bitstrings to integers* on page AppxI-14 to convert the bitstring contents of the instruction operands to the unbounded integers that they represent as two's complement or unsigned integers.
  2. Use of mathematical operators, built-in functions and helper functions on those unbounded integers to calculate other such integers.
  3. Use of either the bitstring extraction operator defined in *Bitstring extraction* on page AppxI-12 or of the saturation helper functions described in *Pseudocode details of saturation* on page A2-9 to convert an unbounded integer result into a bitstring result that can be written to a register.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A2.2.1 整数演算

命令セットには、ビット単位の論理演算、シフト、加算、減算、乗算、その他のレジスタの値に関する幅広い演算が用意されています。これらの演算は、付録 I *「擬似コードの定義」*で説明されている*擬似コード*を使用して、一般に次の 3 つのうちいずれかの方法で定義されます。

- *演算子と組込み関数* P. 付録 I- 11 で定義されている擬似コード演算子およびエンベデッド関数を直接使用する。
- メインテキストで定義されている擬似コードヘルパ関数を使用する。これらは、付録 J *「擬似コード索引」*の表に記載されています。
- 次の形式のシーケンスを使用する。
  1. *ビットストリングの整数への変換* P. 付録 I- 14 で定義されている `SInt()`、`UInt()`、`Int()` エンベデッド関数を使用して、命令オペランドのビットストリングの内容を、2 の補数または符号なし整数を表す、制限なしの整数に変換する。
  2. 得られた制限なしの整数に対して、数学演算子、エンベデッド関数、およびヘルパ関数を適用し、他の整数を計算する。
  3. *ビットストリング抽出* P. 付録 I- 12 で定義されているビットストリング抽出演算子、または*飽和の擬似コードの詳細* P. A2- 9 に記述されている飽和ヘルパ関数を使用して、制限なしの整数の結果をビットストリングに変換し、レジスタに書き込めるようにする。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Shift and rotate operations

The following types of shift and rotate operations are used in instructions:

Logical Shift Left

(LSL) moves each bit of a bitstring left by a specified number of bits. Zeros are shifted in at the right end of the bitstring. Bits that are shifted off the left end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Logical Shift Right

(LSR) moves each bit of a bitstring right by a specified number of bits. Zeros are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Arithmetic Shift Right

(ASR) moves each bit of a bitstring right by a specified number of bits. Copies of the leftmost bit are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Rotate Right (ROR) moves each bit of a bitstring right by a specified number of bits. Each bit that is shifted off the right end of the bitstring is re-introduced at the left end. The last bit shifted off the right end of the bitstring can be produced as a carry output.

Rotate Right with Extend

(RRX) moves each bit of a bitstring right by one bit. The carry input is shifted in at the left end of the bitstring. The bit shifted off the right end of the bitstring can be produced as a carry output.

Pseudocode details of shift and rotate operations

These shift and rotate operations are supported in pseudocode by the following functions:

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);

// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

シフトおよびローテート演算

命令内では、次のタイプのシフトおよびローテート演算が使用されます。

論理左シフト

(LSL) は、ビットストリングの各ビットを指定されたビット数だけ左に移動します。ビットストリングの右端には、0 がシフトインされます。ビットストリングの左端よりもさらに左にシフトされたビットは破棄されますが、その最後のビットをキャリーに出力できます。

論理右シフト

(LSR) は、ビットストリングの各ビットを指定されたビット数だけ右に移動します。ビットストリングの左端には、0 がシフトインされます。ビットストリングの右端よりもさらに右にシフトされたビットは破棄されますが、その最後のビットをキャリーに出力できます。

算術右シフト

(ASR) は、ビットストリングの各ビットを指定されたビット数だけ右に移動します。最も左のビットがコピーされ、ビットストリングの左端にシフトインされていきます。ビットストリングの右端よりもさらに右にシフトされたビットは破棄されますが、その最後のビットをキャリーに出力できます。

右ローテート

(ROR) は、ビットストリングの各ビットを指定されたビット数だけ右に移動します。ビットストリングの右端よりもさらに右にシフトされた各ビットは、左端に再度挿入されます。ビットストリングの右端よりもさらに右にシフトされた最後のビットをキャリーに出力できます。

拡張付き右ローテート

(RRX) は、ビットストリングの各ビットを 1 ビットだけ右に移動します。ビットストリングの左端には、キャリー入力がシフトインされます。ビットストリングの右端よりもさらに右にシフトされたビットを、キャリーに出力できます。

シフトおよびローテート演算の擬似コードの詳細

これらのシフトおよびローテート演算は、擬似コードでは次のような関数として表現されます。

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x :Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);

// LSL()
// =====
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        (result, -) = LSL_C(x, shift);
    return result;

// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;

// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;

// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;

// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;

// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;

// ROR_C()
// =======
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    if n == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;

// RRX_C()
// =======

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);

// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, shift);
    return result;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);

// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    if n == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;

// RRX_C()
// =======

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in :x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);

// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, shift);
    return result;
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Pseudocode details of addition and subtraction

In pseudocode, addition and subtraction can be performed on any combination of unbounded integers and bitstrings, provided that if they are performed on two bitstrings, the bitstrings must be identical in length. The result is another unbounded integer if both operands are unbounded integers, and a bitstring of the same length as the bitstring operand(s) otherwise. For the precise definition of these operations, see *Addition and subtraction* on page AppxI-15.

The main addition and subtraction instructions can produce status information about both unsigned carry and signed overflow conditions. This status information can be used to synthesize multi-word additions and subtractions. In pseudocode the `AddWithCarry()` function provides an addition with a carry input and carry and overflow outputs:

```
// AddWithCarry()
// ==============

(bits(N), bit, bit) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    signed_sum   = SInt(x) + SInt(y) + UInt(carry_in);
    result       = unsigned_sum<N-1:0>;  // == signed_sum<N-1:0>
    carry_out    = if UInt(result) == unsigned_sum then '0' else '1';
    overflow     = if SInt(result) == signed_sum then '0' else '1';
    return (result, carry_out, overflow);
```

An important property of the `AddWithCarry()` function is that if:

```
(result, carry_out, overflow) = AddWithCarry(x, NOT(y), carry_in)
```

then:

- if `carry_in == '1'`, then `result == x-y` with:
  - `overflow == '1'` if signed overflow occurred during the subtraction
  - `carry_out == '1'` if unsigned borrow did not occur during the subtraction, that is, if `x >= y`
- if `carry_in == '0'`, then `result == x-y-1` with:
  - `overflow == '1'` if signed overflow occurred during the subtraction
  - `carry_out == '1'` if unsigned borrow did not occur during the subtraction, that is, if `x > y`.

Together, these mean that the `carry_in` and `carry_out` bits in `AddWithCarry()` calls can act as *NOT borrow* flags for subtractions as well as *carry* flags for additions.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

加算および減算の擬似コードの詳細

擬似コードにおいては、加算および減算は制限なしの整数とビットストリングの任意の組み合わせに対して実行可能ですが、2 つのビットストリングに対して実行される場合は、両方のビットストリングが同じ長さの必要があります。結果は、両方のオペランドが制限なしの整数の場合は別の制限なしの整数、それ以外の場合はビットストリングオペランドと同じ長さのビットストリングになります。これらの演算の正確な定義については、*加算と減算* P. 付録 I-15 を参照して下さい。

主な加算および減算命令では、符号なしキャリーおよび符号付きオーバフロー条件の両方のステータス情報を生成することができます。このステータス情報を使用して、複数ワードの加算および減算を合成できます。次に示す擬似コードの `AddWithCarry()` 関数は、キャリー入力付きの加算を行い、キャリーとオーバフローを出力する例を示しています。

```
// AddWithCarry()
// ==============

(bits(N), bit, bit) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    signed_sum   = SInt(x) + SInt(y) + UInt(carry_in);
    result       = unsigned_sum<N-1:0>;  // == signed_sum<N-1:0>
    carry_out    = if UInt(result) == unsigned_sum then '0' else '1';
    overflow     = if SInt(result) == signed_sum then '0' else '1';
    return (result, carry_out, overflow);
```

`AddWithCarry()` 関数の重要な特徴は、次の条件が満たされる場合に

```
(result, carry_out, overflow) = AddWithCarry(x, NOT(y), carry_in)
```

次の動作を行うことです。

- if `carry_in == '1'` の場合、`result == x-y` で、
  - `overflow == '1'` (減算中に符号付きオーバフローが発生した場合)
  - `carry_out == '1'` (減算中に符号なしボローが発生しなかった、つまり `x >= y` の場合)
- if `carry_in == '0'` の場合、`result == x-y-1` で、
  - `overflow == '1'` (減算中に符号付きオーバフローが発生した場合)
  - `carry_out == '1'` (減算中に符号なしボローが発生しなかった、つまり `x > y` の場合)

つまり、`AddWithCarry()` 呼び出しの `carry_in` および `carry_out` ビットは、減算に対する *NOT borrow* フラグおよび加算に対する *carry* フラグとして使用できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Pseudocode details of saturation

Some instructions perform *saturating arithmetic*, that is, if the result of the arithmetic overflows the destination signed or unsigned N-bit integer range, the result produced is the largest or smallest value in that range, rather than wrapping around modulo 2^N. This is supported in pseudocode by the `SignedSatQ()` and `UnsignedSatQ()` functions when a boolean result is wanted saying whether saturation occurred, and by the `SignedSat()` and `UnsignedSat()` functions when only the saturated result is wanted:

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);

// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);

// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;

// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

`SatQ(i, N, unsigned)` returns either `UnsignedSatQ(i,N)` or `SignedSatQ(i, N)` depending on the value of its third argument, and `Sat(i, N, unsigned)` returns either `UnsignedSat(i, N)` or `SignedSat(i, N)` depending on the value of its third argument:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

飽和の擬似コードの詳細

一部の命令は*飽和演算*を実行します。つまり、演算の結果がデスティネーションの符号付きまたは符号なし N ビット整数の範囲をオーバフローする場合、生成された結果はモジュロ 2^N にラップアラウンドせずに、その範囲内の最大値または最小値になります。演算で飽和が発生したかどうかをブール値として知りたい場合は擬似コードの `SignedSatQ()` および `UnsignedSatQ()` 関数として、飽和した結果のみが必要な場合は `SignedSat()` および `UnsignedSat()` 関数として表されます。


```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);

// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);

// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;

// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```


`SatQ(i, N, unsigned)` は、3 番目の引数の値に応じて `UnsignedSatQ(i, N)` または `SignedSatQ(i, N)` のいずれかを返し、`Sat(i, N, unsigned)` は、3 番目の引数の値に応じて `UnsignedSat(i, N)` または `SignedSat(i, N)` のいずれかを返します。これらの関数を次に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);

// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳


```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);

// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.3 ARM core registers

In the application level view, an ARM processor has:

- thirteen general-purpose32-bit registers, R0 to R12
- three 32-bit registers, R13 to R15, that sometimes or always have a special use.

Registers R13 to R15 are usually referred to by names that indicate their special uses:

SP, the Stack Pointer

Register R13 is used as a pointer to the active stack.

In Thumb code, most instructions cannot access SP. The only instructions that can access SP are those designed to use SP as a stack pointer.

The use of SP for any purpose other than as a stack pointer is deprecated.

Note

Using SP for any purpose other than as a stack pointer is likely to break the requirements of operating systems, debuggers, and other software systems, causing them to malfunction.

LR, the Link Register

Register R14 is used to store the return address from a subroutine. At other times, LR can be used for other purposes.

When a `BL` or `BLX` instruction performs a subroutine call, LR is set to the subroutine return address. To perform a subroutine return, copy LR back to the program counter. This is typically done in one of two ways, after entering the subroutine with a `BL` or `BLX` instruction:

- Return with a `BX LR` instruction.
- On subroutine entry, store LR to the stack with an instruction of the form:
  `PUSH {<registers>,LR}`
  and use a matching instruction to return:
  `POP {<registers>,PC}`

ThumbEE checks and handler calls use LR in a similar way. For details see Chapter A9 *ThumbEE*.

PC, the Program Counter

Register R15 is the program counter:

- When executing an ARM instruction, PC reads as the address of the current instruction plus 8.
- When executing a Thumb instruction, PC reads as the address of the current instruction plus 4.
- Writing an address to PC causes a branch to that address.

In Thumb code, most instructions cannot access PC.

See *ARM core registers* on page B1-9 for the system level view of SP, LR, and PC.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.3 ARM コアレジスタ

アプリケーションレベルのビューでは、ARM プロセッサには次のレジスタが存在します。

- 13 個の汎用 32 ビットレジスタ、R0 ～ R12
- 3 個の 32 ビットレジスタ、R13 ～ R15。これらは場合に応じて、または常に特殊な用途に使用されます。

レジスタ R13 ～ R15 は、通常、その特殊な用途を示す名前で呼ばれます。

SP、スタックポインタ

レジスタ R13 は、アクティブなスタックへのポインタとして使用されます。

Thumb コードでは、ほとんどの命令が SP にアクセスできません。SP にアクセスできる命令は、SP をスタックポインタとして使用するよう意図された命令だけです。

スタックポインタ以外の目的に SP を使用することは推奨されません。

注

SP をスタックポインタ以外の目的に使用すると、オペレーティングシステム、デバッガ、その他のソフトウェアシステムの条件に違反し、それらの誤動作を引き起こすことがあります。

LR、リンクレジスタ

レジスタ R14 は、サブルーチンからの復帰アドレスをストアするために使用されます。それがないときは、他の目的に LR を使用できます。

`BL` または `BLX` 命令がサブルーチン呼び出しを実行する際、LR はサブルーチン復帰アドレスに設定されます。サブルーチン復帰を実行するには、LR を再びプログラムカウンタにコピーします。これは通常、`BL` または `BLX` 命令でサブルーチンに入った後、次の 2 つのうちいずれかの方法で行われます。

- `BX LR` 命令で復帰する。
- サブルーチン開始時に、次の形式の命令で LR をスタックにストアする。
  `PUSH {<registers>,LR}`
  復帰時には、対応する次の命令を使用する。
  `POP {<registers>,PC}`

ThumbEE チェックとハンドラ呼び出しは、LR を同様の形で使用します。詳細については、第 A9 章 *「ThumbEE」* を参照して下さい。

PC、プログラムカウンタ

レジスタ R15 はプログラムカウンタ (PC) です。

- ARM 命令の実行時、PC の読み出し値は現在の命令のアドレス ＋ 8 です。
- Thumb 命令の実行時、PC の読み出し値は現在の命令のアドレス ＋ 4 です。
- アドレスを PC に書き込むと、そのアドレスへの分岐が発生します。

Thumb コードでは、ほとんどの命令が PC にアクセスできません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

The names SP, LR and PC are preferred to R13, R14 and R15. However, sometimes it is simpler to use the R13-R15 names when referring to a group of registers. For example, it is simpler to refer to *Registers R8 to R15*, rather than to *Registers R8 to R12, the SP, LR and PC*. However these two descriptions of the group of registers have exactly the same meaning.

A2.3.1 Pseudocode details of operations on ARM core registers

In pseudocode, the `R[]` function is used to:

- Read or write R0-R12, SP, and LR, using n == 0-12, 13, and 14 respectively.
- Read the PC, using n == 15.

This function has prototypes:

```
bits(32) R[integer n]
    assert n >= 0 && n <= 15;

R[integer n] = bits(32) value
    assert n >= 0 && n <= 14;
```

The full operation of this function is explained in *Pseudocode details of ARM core register operations* on page B1-12.

Descriptions of ARM store instructions that store the PC value use the `PCStoreValue()` pseudocode function to specify the PC value stored by the instruction:

```
// PCStoreValue()
// ==============

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before ARMv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe ARM instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;
```

Writing an address to the PC causes either a simple branch to that address or an *interworking* branch that also selects the instruction set to execute after the branch. A simple branch is performed by the `BranchWritePC()` function:

```
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_ARM then
        if ArchVersion() < 6 && address<1:0> != '00' then UNPREDICTABLE;
        BranchTo(address<31:2>:'00');
    else
        BranchTo(address<31:1>:'0');
```

An interworking branch is performed by the `BXWritePC()` function:

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

SP、LR、PC のシステムレベルのビューについては、*ARM コアレジスタ* P. B1-9 を参照して下さい。

注

R13、R14、R15 よりも SP、LR、PC という名称がよく使用されます。ただし、レジスタのグループを示す場合は、R13 ～ R15 という名称を使用したほうが簡単です。例えば、*レジスタ R8 ～ R12 と SP、LR、PC* よりも、*レジスタ R8 ～ R15* と呼ぶほうが簡単です。しかし、このレジスタのグループについてどちらの呼び方をしても、意味は全く同じです。

A2.3.1 ARM コアレジスタ演算の擬似コードの詳細

擬似コードでは、`R[]` 関数は次の目的に使用されます。

- R0 ～ R12、SP、LR の読み出し / 書き込み。それぞれ n == 0 ～ 12、13、14 を使用します。
- PC の読み出し。n == 15 を使用します。

この関数には、次のようなプロトタイプがあります。

```
bits(32) R[integer n]
    assert n >= 0 && n <= 15;

R[integer n] = bits(32) value
    assert n >= 0 && n <= 14;
```

この関数の完全な動作については、*ARM コアレジスタ演算の擬似コードの詳細* P. B1-12 を参照して下さい。

`PCStoreValue()` 擬似コード関数を使用して命令がストアする PC 値を指定し、PC の値をストアする ARM ストア命令の記述は、次のようになります。

```
// PCStoreValue()
// ==============

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before ARMv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe ARM instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;
```

PC にアドレスを書き込むと、そのアドレスへの単純な分岐が行われるか、*インターワーキング*分岐が行われ、分岐後に実行する命令セットが選択されます。単純な分岐は、`BranchWritePC()` 関数によって実行されます。

```
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_ARM then
        if ArchVersion() < 6 && address<1:0> != '00' then UNPREDICTABLE;
        BranchTo(address<31:2>:'00');
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
// BXWritePC()
// ===========

BXWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_ThumbEE then
        if address<0> == '1' then
            BranchTo(address<31:1>:'0');  // Remaining in ThumbEE state
        else
            UNPREDICTABLE;
    else
        if address<0> == '1' then
            SelectInstrSet(InstrSet_Thumb);
            BranchTo(address<31:1>:'0');
        elsif address<1> == '0' then
            SelectInstrSet(InstrSet_ARM);
            BranchTo(address);
        else // address<1:0> == '10'
            UNPREDICTABLE;
```

The `LoadWritePC()` and `ALUWritePC()` functions are used for two cases where the behavior was systematically modified between architecture versions:

```
// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    if ArchVersion() >= 5 then
        BXWritePC(address);
    else
        BranchWritePC(address);

// ALUWritePC()
// ============

ALUWritePC(bits(32) address)
    if ArchVersion() >= 7 && CurrentInstrSet() == InstrSet_ARM then
        BXWritePC(address);
    else
        BranchWritePC(address);
```

Note

The behavior of the PC writes performed by the `ALUWritePC()` function is different in Debug state, where there are more UNPREDICTABLE cases. The pseudocode in this section only handles the non-debug cases. For more information, see *Data-processing instructions with the PC as the target in Debug state* on page C5-12.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
    else
        BranchTo(address<31:1>:'0');
```

インターワーキング分岐は、`BXWritePC()` 関数によって実行されます。

```
// BXWritePC()
// ===========

BXWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_ThumbEE then
        if address<0> == '1' then
            BranchTo(address<31:1>:'0');  // Remaining in ThumbEE state
        else
            UNPREDICTABLE;
    else
        if address<0> == '1' then
            SelectInstrSet(InstrSet_Thumb) ;
            BranchTo(address<31:1>:'0');
        elsif address<1> == '0' then
            SelectInstrSet(InstrSet_ARM);
            BranchTo(address);
        else // address<1:0> == '10'
            UNPREDICTABLE;
```

`LoadWritePC()` および `ALUWritePC()` 関数は、2 つの場合に使用され、これらの場合には動作がアーキテクチャのバージョンに応じてシステマチックに変更されます。

```
// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    if ArchVersion() >= 5 then
        BXWritePC(address) ;
    else
        BranchWritePC(address) ;

// ALUWritePC()
// ============

ALUWritePC(bits(32) address)
    if ArchVersion() >= 7 && CurrentInstrSet() == InstrSet_ARM then
        BXWritePC(address) ;
    else
        BranchWritePC(address) ;
```

注

`ALUWritePC()` 関数が実行する PC 書き込みの動作は、デバッグ状態では異なります。デバッグ状態では、多くの場合予測不能です。このセクションの擬似コードは、非デバッグ状態の処理のみを行うものです。詳細については、*PC をターゲットとするデータ処理命令のデバッグ状態での動作* P. C5-12 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.4 The Application Program Status Register (APSR)

Program status is reported in the 32-bit *Application Program Status Register* (APSR). The format of the APSR is:

31	30	29	28	27	26 24	23 20	19 16	15 0
N	Z	C	V	Q	RAZ/ SBZP	Reserved	GE[3:0]	Reserved

In the APSR, the bits are in the following categories:

- Reserved bits are allocated to system features, or are available for future expansion. Unprivileged execution ignores writes to privileged fields. However, application level software that writes to the APSR must treat reserved bits as Do-Not-Modify (DNM) bits. For more information about the reserved bits, see *Format of the CPSR and SPSRs* on page B1-16.
- Flags that can be set by many instructions:

  N, bit [31] Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N == 1 if the result is negative and N == 0 if it is positive or zero.

  Z, bit [30] Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

  C, bit [29] Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

  V, bit [28] Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

  Q, bit [27] Set to 1 to indicate overflow or saturation occurred in some instructions, normally related to *Digital Signal Processing* (DSP). For more information, see *Pseudocode details of saturation* on page A2-9.

  GE[3:0], bits [19:16]
  Greater than or Equal flags. SIMD instructions update these flags to indicate the results from individual bytes or halfwords of the operation. These flags can control a later `SEL` instruction. For more information, see *SEL* on page A8-312.
- Bits [26:24] are RAZ/SBZP. Therefore, software can use `MSR` instructions that write the top byte of the APSR without using a read, modify, write sequence. If it does this, it must write zeros to bits [26:24].

Instructions can test the N, Z, C, and V condition code flags to determine whether the instruction is to be executed. In this way, execution of the instruction can be made conditional on the result of a previous operation. For more information about conditional execution see *Conditional execution* on page A4-3 and *Conditional execution* on page A8-8.

In ARMv7-A and ARMv7-R, the APSR is the same register as the CPSR, but the APSR must be used only to access the N, Z, C, V, Q, and GE[3:0] bits. For more information, see *Program Status Registers (PSRs)* on page B1-14.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A2.4 アプリケーションプログラム ステータスレジスタ (APSR)

プログラムステータスは、32 ビットの *アプリケーションプログラム ステータスレジスタ* (APSR) で報告されます。APSR のフォーマットは次のとおりです。

31	30	29	28	27	26 24	23 20	19 16	15 0
N	Z	C	V	Q	RAZ/ SBZP	予約	GE[3:0]	予約

APSR のビットは次のカテゴリに分けられます。

- 予約ビットはシステム機能に割り当てられるか、または将来の拡張の際に使用可能です。非特権実行では、特権フィールドへの書き込みは無視されます。しかし、APSR に書き込みを行うアプリケーションレベルのソフトウェアは、予約ビットを変更不可（DNM）ビットとして扱う必要があります。予約ビットの詳細については、*CPSR と SPSR のフォーマット* P. B1-16 を参照して下さい。
- 次の各ビットは、多くの命令によりフラグとして設定されます。

  N, ビット [31] 負の条件コードフラグ。命令の演算結果のビット [31] がこのフラグに設定されます。演算結果が 2 の補数で符号付き整数の場合、結果が負の場合は N == 1、正またはゼロの場合は N == 0 に設定されます。

  Z, ビット [30] 0 の条件コードフラグ。命令の演算結果が 0 の場合は 1 にセットされ、それ以外の場合は 0 にセットされます。結果が 0 の場合、一般には比較の結果が等しいことを示しています。

  C, ビット [29] キャリーの条件コードフラグ。命令の演算結果でキャリーが出力された場合は 1 にセットされます。例えば、加算での符号なしオーバフローが該当します。

  V, ビット [28] オーバフローの条件コードフラグ。命令の結果がオーバフロー条件の場合は 1 にセットされます。例えば、加算での符号付きオーバフローが該当します。

  Q, ビット [27] 通常は *デジタル信号処理* (DSP) に関連して、一部の命令で発生するオーバフローまたは飽和を示す場合、1 にセットされます。詳細については、*飽和の擬似コードの詳細* P. A2-9 を参照して下さい。

  GE[3:0]、ビット [19:16]
  ≧フラグ。SIMD 命令は、演算の各バイトまたはハーフワードの結果を示すためにこれらのフラグを更新します。これらのフラグは、後の `SEL` 命令で制御可能です。詳細については、*SEL* P. A8-312 を参照して下さい。
- ビット [26:24] は RAZ/SBZP です。したがって、ソフトウェアは `MSR` 命令を使用して、読み出し－変更－書き込みシーケンスを使わずに、APSR の上位バイトに書き込むことができます。これを行う場合には、必ずビット [26:24] に 0 を書き込みます。

命令を実行するかどうかを決定するために、命令で N、Z、C、V の条件コードフラグをテストすることができます。これによって、前の演算の結果が指定の条件を満たしたときのみ、命令を実行することができます。条件付き実行の詳細については、*条件付き実行* P. A4-3 および *条件付き実行* P. A8-8 を参照して下さい。

ARMv7-A と ARMv7-R では、APSR は CPSR と同じレジスタですが、APSR は N、Z、C、V、Q、GE[3:0] ビットにアクセスする場合にのみ使用できます。詳細については、*プログラムステータス レジスタ (PSR)* P. B1-14 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.5 Execution state registers

The execution state registers modify the execution of instructions. They control:

- Whether instructions are interpreted as Thumb instructions, ARM instructions, ThumbEE instructions, or Java bytecodes. For more information, see *ISETSTATE*.
- In Thumb state and ThumbEE state only, what conditions apply to the next four instructions. For more information, see *ITSTATE* on page A2-17.
- Whether data is interpreted as big-endian or little-endian. For more information, see *ENDIANSTATE* on page A2-19.

In ARMv7-A and ARMv7-R, the execution state registers are part of the Current Program Status Register. For more information, see *Program Status Registers (PSRs)* on page B1-14.

There is no direct access to the execution state registers from application level instructions, but they can be changed by side effects of application level instructions.

A2.5.1 ISETSTATE

1	0
J	T

The J bit and the T bit determine the instruction set used by the processor. Table A2-1 shows the encoding of these bits.

Table A2-1 J and T bit encoding in ISETSTATE

J	T	Instruction set state
0	0	ARM
0	1	Thumb
1	0	Jazelle
1	1	ThumbEE

ARM state The processor executes the ARM instruction set described in Chapter A5 *ARM Instruction Set Encoding*.

Thumb state The processor executes the Thumb instruction set as described in Chapter A6 *Thumb Instruction Set Encoding*.

Jazelle state The processor executes Java bytecodes as part of a *Java Virtual Machine* (JVM). For more information, see *Jazelle direct bytecode execution support* on page A2-73.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.5 実行状態レジスタ

実行状態レジスタは、命令の実行を変更します。このレジスタは、次のものを制御します。

- Thumb 命令、ARM 命令、ThumbEE 命令、Java バイトコードのどれに命令を解釈するか。詳細については、*ISETSTATE* を参照して下さい。
- Thumb 状態と ThumbEE 状態の場合のみ、次の 4 つの命令にどの条件を適用するか。詳細については、*ITSTATE* P. A2-17 を参照して下さい。
- ビッグエンディアンとリトルエンディアンのどちらでデータを解釈するか。詳細については、*ENDIANSTATE* P. A2-19 を参照して下さい。

ARMv7-A と ARMv7-R では、実行状態レジスタは、カレントプログラム ステータスレジスタの一部です。詳細については、*プログラム ステータス レジスタ (PSR)* P. B1-14 を参照して下さい。

アプリケーションレベルの命令から実行状態レジスタへの直接アクセスはありませんが、アプリケーションレベル命令の副作用として実行状態レジスタが変更されることがあります。

A2.5.1 ISETSTATE

1	0
J	T

J ビットと T ビットは、プロセッサが使用する命令セットを決定します。これらのビットのエンコードを、表 A2-1 に示します。

表 A2-1 ISETSTATE における J ビットと T ビットのエンコード

J	T	命令セット状態
0	0	ARM
0	1	Thumb
1	0	Jazelle
1	1	ThumbEE

ARM 状態 プロセッサは、第 A5 章 *「ARM 命令セットのエンコード」* に説明されている ARM 命令セットを実行します。

Thumb 状態 プロセッサは、第 A6 章 *「Thumb 命令セットのエンコード」* に説明されている Thumb 命令セットを実行します。

Jazelle 状態 プロセッサは、*Java 仮想マシン* (JVM) の一部として Java バイトコードを実行します。詳細については、*Jazelle 直接バイトコード実行のサポート* P. A2-73 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ThumbEE state The processor executes a variation of the Thumb instruction set specifically targeted for use with dynamic compilation techniques associated with an execution environment. This can be Java or other execution environments. This feature is required in ARMv7-A, and optional in ARMv7-R. For more information, see *Thumb Execution Environment* on page A2-69.

Pseudocode details of ISETSTATE operations

The following pseudocode functions return the current instruction set and select a new instruction set:

```
enumeration InstrSet {InstrSet_ARM, InstrSet_Thumb, InstrSet_Jazelle, InstrSet_ThumbEE};

// CurrentInstrSet()
// =================

InstrSet CurrentInstrSet()
    case ISETSTATE of
        when '00'  result = InstrSet_ARM;
        when '01'  result = InstrSet_Thumb;
        when '10'  result = InstrSet_Jazelle;
        when '11'  result = InstrSet_ThumbEE;
    return result;

// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
    case iset of
        when InstrSet_ARM
            if CurrentInstrSet() == InstrSet_ThumbEE then
                UNPREDICTABLE;
            else
                ISETSTATE = '00';
        when InstrSet_Thumb
            ISETSTATE = '01';
        when InstrSet_Jazelle
            ISETSTATE = '10';
        when InstrSet_ThumbEE
            ISETSTATE = '11';
    return;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ThumbEE 状態 プロセッサは、実行環境に結び付けられている動的なコンパイル手法との併用を特にターゲットとした Thumb 命令セットのバリエーションを実行します。この実行環境は Java のことも、別の環境のこともあります。この機能は ARMv7-A では必須ですが、ARMv7-R ではオプションです。詳細については、*Thumb 実行環境* P. A2-69 を参照して下さい。

ISETSTATE 動作の擬似コードの詳細

次の擬似コード関数は、現在の命令セットを返し、新しい命令セットを選択します。


```
enumeration InstrSet {InstrSet_ARM, InstrSet_Thumb, InstrSet_Jazelle, InstrSet_ThumbEE};

// CurrentInstrSet()
// =================

InstrSet CurrentInstrSet()
    case ISETSTATE of
        when '00'  result = InstrSet_ARM;
        when '01'  result = InstrSet_Thumb;
        when '10'  result = InstrSet_Jazelle;
        when '11'  result = InstrSet_ThumbEE;
    return result;

// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
    case iset of
        when InstrSet_ARM
            if CurrentInstrSet() == InstrSet_ThumbEE then
                UNPREDICTABLE;
            else
                ISETSTATE = '00';
        when InstrSet_Thumb
            ISETSTATE = '01';
        when InstrSet_Jazelle
            ISETSTATE = '10';
        when InstrSet_ThumbEE
            ISETSTATE = '11';
    return;
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.5.2 ITSTATE

7	6	5	4	3	2	1	0
IT[7:0]							

This field holds the If-Then execution state bits for the Thumb IT instruction. See *IT* on page A8-104 for a description of the IT instruction and the associated IT block.

ITSTATE divides into two subfields:

IT[7:5] Holds the *base condition* for the current IT block. The base condition is the top 3 bits of the condition specified by the IT instruction.

This subfield is 0b000 when no IT block is active.

IT[4:0] Encodes:

- The size of the IT block. This is the number of instructions that are to be conditionally executed. The size of the block is implied by the position of the least significant 1 in this field, as shown in Table A2-2 on page A2-18.
- The value of the least significant bit of the condition code for each instruction in the block.

  Note

  Changing the value of the least significant bit of a condition code from 0 to 1 has the effect of inverting the condition code.

This subfield is 0b00000 when no IT block is active.

When an IT instruction is executed, these bits are set according to the condition in the instruction, and the *Then* and *Else* (T and E) parameters in the instruction. For more information, see *IT* on page A8-104.

An instruction in an IT block is conditional, see *Conditional instructions* on page A4-4 and *Conditional execution* on page A8-8. The condition used is the current value of IT[7:4]. When an instruction in an IT block completes its execution normally, ITSTATE is advanced to the next line of Table A2-2 on page A2-18.

For details of what happens if such an instruction takes an exception see *Exception entry* on page B1-34.

Note

Instructions that can complete their normal execution by branching are only permitted in an IT block as its last instruction, and so always result in ITSTATE advancing to normal execution.

Note

ITSTATE affects instruction execution only in Thumb and ThumbEE states. In ARM and Jazelle states, ITSTATE must be '00000000', otherwise behavior is UNPREDICTABLE.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.5.2 ITSTATE

7	6	5	4	3	2	1	0
IT[7:0]							

このフィールドには、Thumb IT 命令の If-Then 実行状態ビットが保持されます。IT 命令と関連する IT ブロックの説明については、*IT* P. A8-104 を参照して下さい。

ITSTATE は、次の 2 つのサブフィールドで構成されます。

IT[7:5] 現在の IT ブロックのベース*条件*が保持されます。これは、IT 命令で指定された条件の上位 3 ビットです。

このサブフィールドは、どの IT ブロックもアクティブでない場合は 0b000 です。

IT[4:0] 次のようにエンコードされます。

- IT ブロックのサイズ。これは、条件付きで実行する命令の数です。P. A2-18 の表 A2-2 に示されているように、ブロックのサイズは、このフィールド内で最下位の 1 の位置によって表されます。
- ブロック内の各命令の条件コード内にある最下位ビットの値

  注

  条件コードの最下位ビットの値が 0 から 1 に変化すると、条件コードが反転します。

このサブフィールドは、どの IT ブロックもアクティブでない場合は 0b00000 です。

IT 命令が実行されると、命令内の条件と、*Then* および *Else*（T および E）パラメータに従って、これらのビットがセットされます。詳細については、*IT* P. A8-104 を参照して下さい。

IT ブロック内部の命令は条件付きです。詳細については、*条件付き命令* P. A4-4 と *条件付き実行* P. A8-8 を参照して下さい。使用される条件は、IT[7:4] の現在の値です。IT ブロック内部の 1 つの命令が正常終了すると、ITSTATE は P. A2-18 の表 A2-2 の次の行に移行します。

このような命令で例外が取得された場合の動作の詳細については、*例外の開始* P. B1-34 を参照して下さい。

注

通常の実行の完了時に分岐する可能性がある命令は、IT ブロック内部の最後の命令にのみ許可されます。これによって、ITSTATE はその命令の実行後に必ず通常の実行に移行します。

注

ITSTATE は、Thumb および ThumbEE 状態でのみ、命令実行に影響を及ぼします。ARM 状態と Jazelle 状態では、ITSTATE は '00000000' の必要があります。そうでない場合、動作は予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A2-2 Effect of IT execution state bits

IT bits [a]						Note
[7:5]	[4]	[3]	[2]	[1]	[0]	
cond_base	P1	P2	P3	P4	1	Entry point for 4-instruction IT block
cond_base	P1	P2	P3	1	0	Entry point for 3-instruction IT block
cond_base	P1	P2	1	0	0	Entry point for 2-instruction IT block
cond_base	P1	1	0	0	0	Entry point for 1-instruction IT block
000	0	0	0	0	0	Normal execution, not in an IT block

a. Combinations of the IT bits not shown in this table are reserved.

Pseudocode details of ITSTATE operations

ITSTATE advances after normal execution of an IT block instruction. This is described by the ITAdvance() pseudocode function:

```
// ITAdvance()
// ===========

ITAdvance()
    if ITSTATE<2:0> == '000' then
        ITSTATE.IT = '00000000';
    else
        ITSTATE.IT<4:0> = LSL(ITSTATE.IT<4:0>, 1);
```

The following functions test whether the current instruction is in an IT block, and whether it is the last instruction of an IT block:

```
// InITBlock()
// ===========

boolean InITBlock()
    return (ITSTATE.IT<3:0> != '0000');

// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (ITSTATE.IT<3:0> == '1000');
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A2- 2 IT 実行状態ビットの効果

IT ビット [a]						注
[7:5]	[4]	[3]	[2]	[1]	[0]	
cond_base	P1	P2	P3	P4	1	4 命令の IT ブロックのエントリポイント
cond_base	P1	P2	P3	1	0	3 命令の IT ブロックのエントリポイント
cond_base	P1	P2	1	0	0	2 命令の IT ブロックのエントリポイント
cond_base	P1	1	0	0	0	1 命令の IT ブロックのエントリポイント
000	0	0	0	0	0	通常の実行、IT ブロック内ではない

a. この表に記載されていない IT ビットの組み合わせは予約されています。

ITSTATE 動作の擬似コードの詳細

ITSTATE は、IT ブロック命令の通常の実行後に更新されます。この動作は、ITAdvance() 擬似コード関数で次のように記述されます。

```
// ITAdvance()
// ===========

ITAdvance()
    if ITSTATE<2:0> == '000' then
        ITSTATE.IT = '00000000';
    else
        ITSTATE.IT<4:0> = LSL(ITSTATE.IT<4:0>, 1);
```

次の関数は、現在の命令が IT ブロック内に存在するかどうかと、その命令が IT ブロックの最後の命令かどうかをテストします。

```
// InITBlock()
// ===========

boolean InITBlock()
    return (ITSTATE.IT<3:0> != '0000');

// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (ITSTATE.IT<3:0> == '1000');
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.5.3 ENDIANSTATE

ARMv7-A and ARMv7-R support configuration between little-endian and big-endian interpretations of data memory, as shown in Table A2-3. The endianness is controlled by ENDIANSTATE.

Table A2-3 APSR configuration of endianness

ENDIANSTATE	Endian mapping
0	Little-endian
1	Big-endian

The ARM and Thumb instruction sets both include an instruction to manipulate ENDIANSTATE:

`SETEND BE` Sets ENDIANSTATE to 1, for big-endian operation

`SETEND LE` Sets ENDIANSTATE to 0, for little-endian operation.

The `SETEND` instruction is unconditional. For more information, see *SETEND* on page A8-314.

Pseudocode details of ENDIANSTATE operations

The `BigEndian()` pseudocode function tests whether big-endian memory accesses are currently selected.

```
// BigEndian()
// ===========

boolean BigEndian()
    return (ENDIANSTATE == '1');
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.5.3 ENDIANSTATE

ARMv7-A と ARMv7-R は、表 A2-3 に示すように、データメモリの解釈についてリトルエンディアンとビッグエンディアンの 2 つの構成をサポートしています。エンディアン形式は、ENDIANSTATE により決定されます。

表 A2-3 APSR によるエンディアンの構成

ENDIANSTATE	エンディアンのマッピング
0	リトルエンディアン
1	ビッグエンディアン

ARM および Thumb 命令セットのどちらにも、ENDIANSTATE を操作する命令が含まれています。

`SETEND BE` ENDIANSTATE を 1 にセットし、ビッグエンディアンの動作を設定します。

`SETEND LE` ENDIANSTATE を 0 にセットし、リトルエンディアンの動作を設定します。

`SETEND` 命令は無条件です。詳細については、*SETEND* P. A8-314 を参照して下さい。

ENDIANSTATE 動作の擬似コードの詳細

`BigEndian()` 擬似コード関数は、ビッグエンディアンのメモリアクセスが現在選択されているかどうかをテストします。

```
// BigEndian()
// ===========

boolean BigEndian()
    return (ENDIANSTATE == '1');
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.6 Advanced SIMD and VFP extensions

Advanced SIMD and VFP are two optional extensions to ARMv7.

Advanced SIMD performs packed *Single Instruction Multiple Data* (SIMD) operations, either integer or single-precision floating-point. VFP performs single-precision or double-precision floating-point operations.

Both extensions permit *floating-point exceptions*, such as overflow or division by zero, to be handled in an untrapped fashion. When handled in this way, a floating-point exception causes a cumulative status register bit to be set to 1 and a default result to be produced by the operation.

The ARMv7 VFP implementation is VFPv3. ARMv7 also permits a variant of VFPv3, VFPv3U, that supports the trapping of floating-point exceptions, see *VFPv3U* on page A2-31. VFPv2 also supports the trapping of floating-point exceptions.

For more information about floating-point exceptions see *Floating-point exceptions* on page A2-42.

Each extension can be implemented at a number of levels. Table A2-4 shows the permitted combinations of implementations of the two extensions.

Table A2-4 Permitted combinations of Advanced SIMD and VFP extensions

Advanced SIMD	VFP
Not implemented	Not implemented
Integer only	Not implemented
Integer and single-precision floating-point	Single-precision floating-point only[a]
Integer and single-precision floating-point	Single-precision and double-precision floating-point
Not implemented	Single-precision floating-point only[a]
Not implemented	Single-precision and double-precision floating-point

a. Must be able to load and store double-precision data.

The optional half-precision extensions provide conversion functions in both directions between half-precision floating-point and single-precision floating-point. These extensions can be implemented with any Advanced SIMD and VFP implementation that supports single-precision floating-point. The half-precision extensions apply to both VFP and Advanced SIMD if they are both implemented.

For system-level information about the Advanced SIMD and VFP extensions see:

- *Advanced SIMD and VFP extension system registers* on page B1-66
- *Advanced SIMD and floating-point support* on page B1-64.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A2.6 アドバンスト SIMD および VFP 拡張機能

アドバンスト SIMD と VFP は、ARMv7 にオプションとして装備される 2 つの拡張機能です。

アドバンスト SIMD は、整数または単精度浮動小数点のパックされた*単一命令複数データ* (SIMD) 演算を実行します。VFP は、単精度または倍精度の浮動小数点演算を実行します。

どちらの拡張機能でも、オーバフロー、0 による除算などの*浮動小数点例外*をトラップせずに処理することが許可されています。このように処理された場合、浮動小数点例外により累積ステータスレジスタ ビットが 1 にセットされ、演算によるデフォルトの結果が生成されます。

ARMv7 の VFP 実装は VFPv3 です。ARMv7 では、VFPv3 のバリアントである VFPv3U も許可されています。このバリアントでは、浮動小数点例外のトラップがサポートされています。*VFPv3U* P. A2-31 を参照して下さい。VFPv2 でも、浮動小数点例外のトラップがサポートされています。

浮動小数点例外の詳細については、*浮動小数点例外* P. A2-42 を参照して下さい。

各拡張命令は多くの異なるレベルで実装可能です。これら 2 つの拡張機能の実装で許可される組み合わせを、表 A2-4 に示します。

表 A2-4 アドバンスト SIMD 拡張機能と VFP 拡張機能の許可されている組み合わせ

アドバンスト SIMD	VFP
実装なし	実装なし
整数のみ	実装なし
整数と単精度浮動小数点	単精度浮動小数点のみ[a]
整数と単精度浮動小数点	単精度と倍精度の浮動小数点
実装なし	単精度浮動小数点のみ[a]
実装なし	単精度と倍精度の浮動小数点

a. 倍精度データをロード / ストアできる必要があります。

オプションの半精度拡張機能は、半精度浮動小数点と単精度浮動小数点とを相互に変換する変換関数を提供します。この拡張機能は、単精度浮動小数点をサポートするどのアドバンスト SIMD および VFP の実装にも実装可能です。半精度拡張機能は、VFP とアドバンスト SIMD の両方が実装されている場合、両方に適用されます。

アドバンスト SIMD 拡張機能と VFP 拡張機能に関するシステムレベルの情報については、次のセクションを参照して下さい。

- *アドバンスト SIMD および VFP 拡張機能のシステムレジスタ* P. B1-66
- *アドバンスト SIMD と浮動小数点のサポート* P. B1-64

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

Before ARMv7, the VFP extension was called the *Vector Floating-point Architecture*, and was used for vector operations. For details of these deprecated operations see Appendix F *VFP Vector Operation Support*. From ARMv7:

- ARM recommends that the Advanced SIMD extension is used for single-precision vector floating-point operations
- an implementation that requires support for vector operations must implement the Advanced SIMD extension.

A2.6.1 Advanced SIMD and VFP extension registers

Advanced SIMD and VFPv3 use the same register set. This is distinct from the ARM core register set. These registers are generally referred to as the *extension registers*.

The extension register set consists of either thirty-two or sixteen doubleword registers, as follows:

- If VFPv2 is implemented, it consists of sixteen doubleword registers.
- If VFPv3 is implemented, it consists of either thirty-two or sixteen doubleword registers. Where necessary the terms VFPv3-D32 and VFPv3-D16 are used to distinguish between these two implementation options.
- If Advanced SIMD is implemented, it consists of thirty-two doubleword registers. If both Advanced SIMD and VFPv3 are implemented, VFPv3 must be implemented in its VFPv3-D32 form.

The Advanced SIMD and VFP views of the extension register set are not identical. They are described in the following sections.

Figure A2-1 on page A2-22 shows the views of the extension register set, and the way the word, doubleword, and quadword registers overlap.

Advanced SIMD views of the extension register set

Advanced SIMD can view this register set as:

- Sixteen 128-bit quadword registers, `Q0-Q15`.
- Thirty-two 64-bit doubleword registers, `D0-D31`. This view is also available in VFPv3.

These views can be used simultaneously. For example, a program might hold 64-bit vectors in D0 and D1 and a 128-bit vector in Q1.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

ARMv7 以前では、VFP 拡張機能は*ベクタ浮動小数点アーキテクチャ*と呼ばれ、ベクタ演算に使用されていました。これらの非推奨演算の詳細については、付録 F *「VFP ベクタ演算のサポート」*を参照して下さい。ARMv7 およびそれ以降では、次の条件が適用されます。

- 単精度ベクタ浮動小数点演算には、アドバンスト SIMD 拡張機能の使用を推奨します。
- ベクタ演算のサポートを必要とする実装は、アドバンスト SIMD 拡張機能を実装する必要があります。

A2.6.1 アドバンスト SIMD と VFP の拡張機能レジスタ

アドバンスト SIMD と VFPv3 は、同じレジスタセットを使用します。これは、ARM コアのレジスタセットとは明確に区別されます。これらのレジスタは一般に、*拡張機能レジスタ*と呼ばれます。

拡張機能レジスタセットは、次のように 32 個または 16 個のダブルワードレジスタで構成されます。

- VFPv2 が実装されている場合、16 個のダブルワードレジスタで構成されます。
- VFPv3 が実装されている場合、32 個または 16 個のいずれかのダブルワードレジスタで構成されます。必要に応じ、VFPv3-D32、VFPv3-D16 という用語を使用してこれらの 2 つの実装オプションを区別します。
- アドバンスト SIMD が実装されている場合、32 個のダブルワードレジスタで構成されます。アドバンスト SIMD と VFPv3 の両方が実装されている場合、VFPv3 は VFPv3-D32 の形で実装する必要があります。

拡張機能レジスタセットのアドバンスト SIMD ビューと VFP ビューは、全く同じではありません。これらについては以下のセクションで紹介します。

拡張機能レジスタセットのビュー、およびワード、ダブルワード、クワッドワードレジスタのオーバラップを、P. A2-23 の図 A2-1 に示します。

拡張機能レジスタセットのアドバンスト SIMD ビュー

アドバンスト SIMD では、このレジスタセットを次のように認識できます。

- 128 ビットのクワッドワードレジスタ 16 個、`Q0-Q15`
- 64 ビットのダブルワードレジスタ 32 個、`D0-31` このビューは、VFPv3 にもあります。

これらのビューは同時に使用できます。例えば、プログラムで、D0 と D1 に 64 ビットベクタ、Q1 に 128 ビットベクタを保持することができます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

VFP views of the extension register set

In VFPv3-D32, the extension register set consists of thirty-two doubleword registers, that VFP can view as:

- Thirty-two 64-bit doubleword registers, `D0-D31`. This view is also available in Advanced SIMD.
- Thirty-two 32-bit single word registers, `S0-S31`. Only half of the set is accessible in this view.

In VFPv3-D16 and VFPv2, the extension register set consists of sixteen doubleword registers, that VFP can view as:

- Sixteen 64-bit doubleword registers, `D0-D15`.
- Thirty-two 32-bit single word registers, `S0-S31`.

In each case, the two views can be used simultaneously.

Advanced SIMD and VFP register mapping

Figure A2-1 Advanced SIMD and VFP register set

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

拡張機能レジスタセットの VFP ビュー

VFPv3-D32 では、拡張機能レジスタセットは 32 個のダブルワードレジスタで構成されます。VFP では、このレジスタセットを次のように認識できます。

- 64 ビットのダブルワードレジスタ 32 個、`D0-D31`。このビューは、アドバンスト SIMD にもあります。
- 32 ビットのシングルワードレジスタ 32 個、`S0-S31`。このビューでは、セットの半分だけにアクセス可能です。

VFPv3-D16 と VFPv2 では、拡張機能レジスタセットは 16 個のダブルワードレジスタで構成されます。VFP では、このレジスタセットを次のように認識できます。

- 64 ビットのダブルワードレジスタ 16 個、`D0-D15`
- 32 ビットのシングルワードレジスタ 32 個、`S0-S31`

どちらの場合も、2 つのビューを同時に使用できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The mapping between the registers is as follows:

- S<2n> maps to the least significant half of D<n>
- S<2n+1> maps to the most significant half of D<n>
- D<2n> maps to the least significant half of Q<n>
- D<2n+1> maps to the most significant half of Q<n>.

For example, you can access the least significant half of the elements of a vector in Q6 by referring to D12, and the most significant half of the elements by referring to D13.

Pseudocode details of Advanced SIMD and VFP extension registers

The pseudocode function VFPSmallRegisterBank() returns FALSE if all of the 32 registers D0-D31 can be accessed, and TRUE if only the 16 registers D0-D15 can be accessed:

```
boolean VFPSmallRegisterBank()
```

In more detail, VFPSmallRegisterBank():

- returns TRUE for a VFPv2 or VFPv3-D16 implementation
- for a VFPv3-D32 implementation:
  - — returns FALSE if CPACR.D32DIS == 0
  - — returns TRUE if CPACR.D32DIS == 1 and CPACR.ASEDIS == 1
  - — results in UNPREDICTABLE behavior if CPACR.D32DIS == 1 and CPACR.ASEDIS == 0.

For details of the CPACR register, see:

- *c1, Coprocessor Access Control Register (CPACR)* on page B3-104 for a VMSA implementation
- *c1, Coprocessor Access Control Register (CPACR)* on page B4-51 for a PMSA implementation.

The S0-S31, D0-D31, and Q0-Q15 views of the registers are provided by the following functions:

```
// The 64-bit extension register bank for Advanced SIMD and VFP.

array bits(64) _D[0..31];

// S[] - non-assignment form
// =========================

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    if (n MOD 2) == 0 then
        result = D[n DIV 2]<31:0>;
    else
        result = D[n DIV 2]<63:32>;
    return result;

// S[] - assignment form
// =====================

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    if (n MOD 2) == 0 then
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アドバンスト SIMD と VFP レジスタのマッピング

図 A2-1 アドバンスト SIMD と VFP のレジスタセット

レジスタ間のマッピングは次のとおりです。

- S<2n> は、D<n> の下位半分にマッピングされます。
- S<2n+1> は、D<n> の上位半分にマッピングされます。
- D<2n> は、Q<n> の下位半分にマッピングされます。
- D<2n+1> は、Q<n> の上位半分にマッピングされます。

例えば、Q6 のベクタのエレメントの下位半分にアクセスするには D12 を参照し、エレメントの上位半分にアクセスするには D13 を参照します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        D[n DIV 2]<31:0> = value;
    else
        D[n DIV 2]<63:32> = value;
    return;

// D[] - non-assignment form
// =========================

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    if n >= 16 && VFPSmallRegisterBank() then UNDEFINED;
    return _D[n];

// D[] - assignment form
// =====================

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    if n >= 16 && VFPSmallRegisterBank() then UNDEFINED;
    _D[n] = value;
    return;

// Q[] - non-assignment form
// =========================

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return D[2*n+1]:D[2*n];

// Q[] - assignment form
// =====================

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    D[2*n] = value<63:0>;
    D[2*n+1] = value<127:64>;
    return;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アドバンスト SIMD と VFP の拡張機能レジスタの擬似コードの詳細

擬似コード関数 `VFPSmallRegisterBank()` は、D0 ～ D31 の 32 個のレジスタすべてにアクセス可能な場合は FALSE を、D0 ～ D15 の 16 個のレジスタだけにアクセスできる場合は TRUE を返します。

```
boolean VFPSmallRegisterBank()
```

`VFPSmallRegisterBank()` の詳細は次のとおりです。

- VFPv2 実装または VFPv3-D16 実装の場合は TRUE を返します。
- VFPv3-D32 実装の場合は、条件によって次の値を返します。
  - CPACR.D32DIS == 0 の場合は、FALSE を返します。
  - CPACR.D32DIS == 1 かつ CPACR.ASEDIS == 1 の場合は、TRUE を返します。
  - CPACR.D32DIS == 1 かつ CPACR.ASEDIS == 0 の場合、予測不能な動作が引き起こされます。

CPACR レジスタの詳細については、次を参照して下さい。

- *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B3-104（VMSA 実装の場合）
- *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B4-51 （PMSA 実装の場合）

レジスタの S0 ～ S31、D0 ～ D31、Q0 ～ Q15 のビューは、次の関数によって提供されます。

```
// The 64-bit extension register bank for Advanced SIMD and VFP.

array bits(64) _D[0..31];

// S[] - non-assignment form
// =========================

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    if (n MOD 2) == 0 then
        result = D[n DIV 2]<31:0>;
    else
        result = D[n DIV 2]<63:32>;
    return result;

// S[] - assignment form
// =====================

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    if (n MOD 2) == 0 then
        D[n DIV 2]<31:0> = value;
    else
        D[n DIV 2]<63:32> = value;
    return;

// D[] - non-assignment form
// =========================

bits(64) D[integer n]
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.6.2 Data types supported by the Advanced SIMD extension

When the Advanced SIMD extension is implemented, it can operate on integer and floating-point data. It defines a set of data types to represent the different data formats. Table A2-5 shows the available formats. Each instruction description specifies the data types that the instruction supports.

Table A2-5 Advanced SIMD data types

Data type specifier	Meaning
.<size>	Any element of <size> bits
.F<size>	Floating-point number of <size> bits
.I<size>	Signed or unsigned integer of <size> bits
.P<size>	Polynomial over {0,1} of degree less than <size>
.S<size>	Signed integer of <size> bits
.U<size>	Unsigned integer of <size> bits

The polynomial data type is described in *Polynomial arithmetic over {0,1}* on page A2-67.

The .F16 data type is the half-precision data type currently selected by the FPSCR.AHP bit, see *Advanced SIMD and VFP system registers* on page A2-28. It is supported only when the half-precision extensions are implemented.

The .F32 data type is the ARM standard single-precision floating-point data type, see *Advanced SIMD and VFP single-precision format* on page A2-34.

The instruction definitions use a data type specifier to define the data types appropriate to the operation. Figure A2-2 on page A2-26 shows the hierarchy of Advanced SIMD data types.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
    assert n >= 0 && n <= 31;
    if n >= 16 && VFPSmallRegisterBank() then UNDEFINED;
    return _D[n];

// D[] - assignment form
// =====================

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    if n >= 16 && VFPSmallRegisterBank() then UNDEFINED;
    _D[n] = value;
    return;

// Q[] - non-assignment form
// =========================

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return D[2*n+1]:D[2*n];

// Q[] - assignment form
// =====================

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    D[2*n] = value<63:0>;
    D[2*n+1] = value<127:64>;
    return;
```

A2.6.2 アドバンスト SIMD 拡張機能でサポートされるデータ型

アドバンスト SIMD 拡張機能は、実装されている場合、整数と浮動小数点のデータに対して演算を実行できます。アドバンスト SIMD 拡張機能は、さまざまなデータフォーマットを表現するデータ型のセットを定義しています。使用できるフォーマットを、表 A2-5 に示します。各命令の説明には、その命令がサポートするデータ型が指定されています。

表 A2-5 アドバンスト SIMD のデータ型

データ型指定子	意味
.<size>	<size> ビットのエレメントすべて
.F<size>	<size> ビットの浮動小数点数
.I<size>	<size> ビットの符号付きまたは符号なし整数
.P<size>	{0,1} における <size> より次数の小さい多項式
.S<size>	<size> ビットの符号付き整数
.U<size>	<size> ビットの符号なし整数

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

‡ Supported only if the half-precision extensions are implemented

Figure A2-2 Advanced SIMD data type hierarchy

For example, a multiply instruction must distinguish between integer and floating-point data types. However, some multiply instructions use modulo arithmetic for integer instructions and therefore do not need to distinguish between signed and unsigned inputs.

A multiply instruction that generates a double-width (long) result must specify the input data types as signed or unsigned, because for this operation it does make a difference.

A2.6.3 Advanced SIMD vectors

When the Advanced SIMD extension is implemented, a register can hold one or more packed elements, all of the same size and type. The combination of a register and a data type describes a vector of elements. The vector is considered to be an array of elements of the data type specified in the instruction. The number of elements in the vector is implied by the size of the data elements and the size of the register.

Vector indices are in the range 0 to (number of elements – 1). An index of 0 refers to the least significant end of the vector. Figure A2-3 on page A2-27 shows examples of Advanced SIMD vectors:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

多項式のデータ型については、*{0,1} における多項式算術演算* P. A2-67 で説明します。

`.F16` のデータ型は、FPSCR.AHP ビットによって現在選択されている半精度のデータ型です。*アドバンスト SIMD と VFP のシステムレジスタ* P. A2-28 を参照して下さい。このデータ型は、半精度拡張機能が実装されている場合のみサポートされます。

`.F32` のデータ型は、ARM の標準的な単精度浮動小数点データ型です。*アドバンスト SIMD と VFP の単精度フォーマット* P. A2-34 を参照して下さい。

命令の定義では、データ型指定子を使用して演算に適切なデータ型を定義しています。アドバンスト SIMD のデータ型の階層を、P. A2-26 の図 A2-2 に示します。

‡ 半精度拡張機能が実装されている場合のみサポートされます。

図 A2-2 アドバンスト SIMD のデータ型の階層

例えば、乗算命令は整数のデータ型と浮動小数点のデータ型を区別する必要があります。しかし、一部の乗算命令は整数命令にモジュロ算術演算を使用しているため、符号付きの入力と符号なしの入力を区別する必要がありません。

ダブル幅（ロング）の結果を生成する乗算命令では、結果に違いが生じるため、入力データ型が符号付きか符号なしかを指定する必要があります。

A2.6.3 アドバンスト SIMD のベクタ

アドバンスト SIMD 拡張機能が実装されている場合、レジスタは 1 つ以上のパックされたエレメントを保持することができます。これらのエレメントはすべて同じサイズと型です。ベクタのエレメントは、レジスタとデータ型の組み合わせにより記述されます。ベクタは、命令で指定されたデータ型のエレメントの配列と見なされます。ベクタのエレメント数は、データエレメントのサイズとレジスタのサイズによって暗黙的に示されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure A2-3 Examples of Advanced SIMD vectors

Pseudocode details of Advanced SIMD vectors

The pseudocode function `Elem[]` is used to access the element of a specified index and size in a vector:

```
// Elem[] - non-assignment form
// ============================

bits(size) Elem[bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    return vector<(e+1)*size-1:e*size>;

// Elem[] - assignment form
// ========================

Elem[bits(N) vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ベクタのインデクスの範囲は、0 ～（エレメント数 - 1）です。インデクス 0 は、ベクタの最下位を意味します。アドバンスト SIMD ベクタの例を、P. A2-27 の図 A2-3 に示します。

図 A2-3 アドバンスト SIMD ベクタの例

アドバンスト SIMD ベクタの擬似コードの詳細

ベクタ内の指定されたインデクスとサイズを持つエレメントにアクセスするには、擬似関数 `Elem[]` を使用します。

```
// Elem[] - non-assignment form
// ============================

bits(size) Elem[bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    return vector<(e+1)*size-1:e*size>;

// Elem[] - assignment form
// ========================

Elem[bits(N) vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.6.4 Advanced SIMD and VFP system registers

The Advanced SIMD and VFP extensions have a shared register space for system registers. Only one register in this space is accessible at the application level, see *Floating-point Status and Control Register (FPSCR)*.

See *Advanced SIMD and VFP extension system registers* on page B1-66 for the system level description of the registers.

Floating-point Status and Control Register (FPSCR)

The Floating-point Status and Control Register (FPSCR) is implemented in any system that implements one or both of:

- the VFP extension
- the Advanced SIMD extension.

The FPSCR provides all necessary User level control of the floating-point system

The FPSCR is a 32-bit read/write system register, accessible in unprivileged and privileged modes.

The format of the FPSCR is:

Bits [31:28] Condition code bits. These are updated on floating-point comparison operations. They are not updated on SIMD operations, and do not affect SIMD instructions.

N, bit [31] Negative condition code flag.

Z, bit [30] Zero condition code flag.

C, bit [29] Carry condition code flag.

V, bit [28] Overflow condition code flag.

QC, bit [27] Cumulative saturation flag, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit. For details of saturation, see *Pseudocode details of saturation* on page A2-9.

The value of this bit is ignored by the VFP extension. If Advanced SIMD is not implemented this bit is UNK/SBZP.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.6.4 アドバンスト SIMD と VFP のシステムレジスタ

アドバンスト SIMD および VFP の拡張機能では、システムレジスタのレジスタ空間が共有されます。アプリケーションレベルでは、この空間のうち 1 つのレジスタにのみアクセス可能です。*浮動小数点ステータスおよび制御レジスタ (FPSCR)* を参照して下さい。

レジスタのシステムレベルでの説明については、*アドバンスト SIMD および VFP 拡張機能のシステムレジスタ* P. B1-66 を参照して下さい。

浮動小数点ステータスおよび制御レジスタ (FPSCR)

浮動小数点ステータスおよび制御レジスタ (FPSCR) は、次のいずれか、または両方を実装するすべてのシステムに実装されています。

- VFP 拡張機能
- アドバンスト SIMD 拡張機能

FPSCR は、浮動小数点システムに必要なすべてのユーザレベル制御を提供します。

FPSCR は、32 ビットの読み出し / 書き込みシステムレジスタで、特権モードと非特権モードの両方でアクセス可能です。

FPSCR のフォーマットは次のとおりです。

ビット [31:28] 条件コードビット。これらのビットは、浮動小数点比較演算の際に更新されます。SIMD 演算では更新されず、SIMD 命令に影響を与えることもありません。

N、ビット [31] 負の条件コードフラグ

Z、ビット [30] 0 の条件コードフラグ

C、ビット [29] キャリーの条件コードフラグ

V、ビット [28] オーバフローの条件コードフラグ

QC、ビット [27] 累積飽和フラグで、アドバンスト SIMD にのみ存在します。このビットが 1 にセットされている場合、このビットに最後に 0 が書き込まれた後で、アドバンスト SIMD 整数演算が飽和したことを示します。詳細については、*飽和の擬似コードの詳細* P. A2-9 を参照して下さい。

VFP 拡張機能では、このビットの値は無視されます。アドバンスト SIMD が実装されていない場合、このビットは UNK/SBZP です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

AHP, bit[26] Alternative half-precision control bit:

0 IEEE half-precision format selected.

1 Alternative half-precision format selected.

For more information see *Advanced SIMD and VFP half-precision formats* on page A2-38.

If the half-precision extensions are not implemented this bit is UNK/SBZP.

Bits [19,14:13,6:5]

Reserved. UNK/SBZP.

DN, bit [25] Default NaN mode control bit:

0 NaN operands propagate through to the output of a floating-point operation.

1 Any operation involving one or more NaNs returns the Default NaN.

For more information, see *NaN handling and the Default NaN* on page A2-41.

The value of this bit only controls VFP arithmetic. Advanced SIMD arithmetic always uses the Default NaN setting, regardless of the value of the DN bit.

FZ, bit [24] Flush-to-zero mode control bit:

0 Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.

1 Flush-to-zero mode enabled.

For more information, see *Flush-to-zero* on page A2-39.

The value of this bit only controls VFP arithmetic. Advanced SIMD arithmetic always uses the Flush-to-zero setting, regardless of the value of the FZ bit.

RMode, bits [23:22]

Rounding Mode control field. The encoding of this field is:

0b00 *Round to Nearest* (RN) mode

0b01 *Round towards Plus Infinity* (RP) mode

0b10 *Round towards Minus Infinity* (RM) mode

0b11 *Round towards Zero* (RZ) mode.

The specified rounding mode is used by almost all VFP floating-point instructions. Advanced SIMD arithmetic always uses the Round to Nearest setting, regardless of the value of the RMode bits.

Stride, bits [21:20] and Len, bits [18:16]

Use of nonzero values of these fields is deprecated in ARMv7. For details of their use in previous versions of the ARM architecture see Appendix F *VFP Vector Operation Support*.

The values of these fields are ignored by the Advanced SIMD extension.

Bits [15,12:8] Floating-point exception trap enable bits. These bits are supported only in VFPv2 and VFPv3U. They are reserved, RAZ/SBZP, on a system that implements VFPv3.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

AHP、ビット [26] 代替半精度の制御ビット。

0 IEEE 半精度フォーマットが選択されています。

1 代替半精度フォーマットが選択されています。

詳細については、*アドバンスト SIMD と VFP の半精度フォーマット* P. A2-38 を参照して下さい。

半精度拡張機能が実装されていない場合、このビットは UNK/SBZP です。

ビット [19、14:13、6:5]

予約。UNK/SBZP。

DN、ビット [25] デフォルト NaN モードの制御ビット。

0 NaN オペランドは浮動小数点演算の出力まで伝播されます。

1 1 つ以上の NaN を含む演算はすべて、デフォルト NaN を返します。

詳細については、*NaN の処理とデフォルト NaN* P. A2-41 を参照して下さい。

このビットの値は、VFP 算術演算だけを制御します。アドバンスト SIMD 算術演算は、DN ビットの値に関係なく常にデフォルト NaN の設定を使用します。

FZ、ビット [24] Flush-to-Zero モード制御ビット。

0 Flush-to-Zero モードが不可能です。浮動小数点システムの動作は、IEEE 754 規格に完全に準拠しています。

1 Flush-to-Zero モードが可能です。

詳細については、*Flush-to-Zero* P. A2-39 を参照して下さい。

このビットの値は、VFP 算術演算だけを制御します。アドバンスト SIMD 算術演算は、FZ ビットの値に関係なく常に Flush-to-Zero の設定を使用します。

RMode、ビット [23:22]

丸めモード制御フィールド。このフィールドのエンコードは次のとおりです。

0b00 *近似値への丸め* (RN) モード

0b01 *プラス無限大への丸め* (RP) モード

0b10 *マイナス無限大への丸め* (RP) モード

0b11 *ゼロへの丸め* (RZ) モード

指定された丸めモードは、ほとんどの VFP 浮動小数点命令によって使用されます。アドバンスト SIMD 算術演算は、RMode ビットの値に関係なく、常に近似値への丸めの設定を使用します。

Stride、ビット [21:20] および Len、ビット [18:16]

これらのフィールドに 0 以外の値を使用することは、ARMv7 では推奨されません。ARM アーキテクチャの以前のバージョンでこれらの値を使用する場合の詳細については、付録 F 「*VFP ベクタ演算のサポート*」を参照して下さい。

アドバンスト SIMD 拡張機能では、これらのフィールドの値は無視されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The possible values of each bit are:

0 Untrapped exception handling selected

1 Trapped exception handling selected.

The values of these bits control only VFP arithmetic. Advanced SIMD arithmetic always uses untrapped exception handling, regardless of the values of these bits.

For more information, see *Floating-point exceptions* on page A2-42.

IDE, bit [15] Input Denormal exception trap enable.

IXE, bit [12] Inexact exception trap enable.

UFE, bit [11] Underflow exception trap enable.

OFE, bit [10] Overflow exception trap enable.

DZE, bit [9] Division by Zero exception trap enable.

IOE, bit [8] Invalid Operation exception trap enable.

Bits [7,4:0] Cumulative exception flags for floating-point exceptions. Each of these bits is set to 1 to indicate that the corresponding exception has occurred since 0 was last written to it. How VFP instructions update these bits depends on the value of the corresponding exception trap enable bits:

Trap enable bit = 0

If the floating-point exception occurs then the cumulative exception flag is set to 1.

Trap enable bit = 1

If the floating-point exception occurs the trap handling software can decide whether to set the cumulative exception flag to 1.

Advanced SIMD instructions set each cumulative exception flag if the corresponding exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the setting of the trap enable bits.

For more information, see *Floating-point exceptions* on page A2-42.

IDC, bit [7] Input Denormal cumulative exception flag.

IXC, bit [4] Inexact cumulative exception flag.

UFC, bit [3] Underflow cumulative exception flag.

OFC, bit [2] Overflow cumulative exception flag.

DZC, bit [1] Division by Zero cumulative exception flag.

IOC, bit [0] Invalid Operation cumulative exception flag.

If the processor implements the integer-only Advanced SIMD extension and does not implement the VFP extension, all of these bits except QC are UNK/SBZP.

Writes to the FPSCR can have side-effects on various aspects of processor operation. All of these side-effects are synchronous to the FPSCR write. This means they are guaranteed not to be visible to earlier instructions in the execution stream, and they are guaranteed to be visible to later instructions in the execution stream.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ビット [15、12:8] 浮動小数点例外トラップイネーブルビット。これらのビットは VFPv2 と VFPv3U でのみサポートされています。VFPv3 を実装しているシステムでは、これらは予約ビットで RAZ/SBZP です。

各ビットの可能な値は次のとおりです。

0 トラップなしの例外処理が選択されています。

1 トラップ例外処理が選択されています。

これらのビットの値は、VFP 算術演算だけを制御します。アドバンスト SIMD 算術演算は、これらのビットの値に関係なく、常にトラップなしの例外処理を使用します。

詳細については、*浮動小数点例外* P. A2- 42 を参照して下さい。

IDE、ビット [15] 入力非正規化例外トラップのイネーブル。

IXE、ビット [12] 不正確例外トラップのイネーブル。

UFE、ビット [11] アンダーフロー例外トラップのイネーブル。

OFE、ビット [10] オーバフロー例外トラップのイネーブル。

DZE、ビット [9] 0 による除算例外トラップのイネーブル。

IOE、ビット [8] 無効演算例外トラップのイネーブル。

ビット [7、4:0] 浮動小数点例外の累積例外フラグ。それぞれのビットが 1 にセットされている場合、そのビットに最後に 0 が書き込まれた後で、対応する例外が発生したことを示しています。VFP 命令でどのようにこれらのビットが更新されるかは、対応する例外トラップ イネーブルビットの値によって決まります。

トラップ イネーブルビット = 0

浮動小数点例外が発生した場合、累積例外フラグは 1 にセットされます。

トラップ イネーブルビット = 1

浮動小数点例外が発生した場合、トラップ処理ソフトウェアが、累積例外フラグを 1 にセットするかどうかを決定できます。

アドバンスト SIMD 命令は、トラップ イネーブルビットの設定に関係なく、命令により実行された浮動小数点計算の 1 つ以上で例外が発生した場合に、対応する累積例外フラグをセットします。

詳細については、*浮動小数点例外* P. A2- 42 を参照して下さい。

IDC、ビット [7] 入力非正規化累積例外フラグ。

IXC、ビット [4] 不正確累積例外フラグ。

UFC、ビット [3] アンダーフロー累積例外フラグ。

OFC、ビット [2] オーバフロー累積例外フラグ。

DZC、ビット [1] 0 による除算累積例外フラグ。

IOC、ビット [0] 無効演算累積例外フラグ。

プロセッサに整数専用のアドバンスト SIMD 拡張機能が実装され、VFP 拡張機能が実装されていない場合、QC 以外のすべてのビットは UNK/SBZP です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the FPSCR

You read or write the FPSCR using the `VMRS` and `VMSR` instructions. For more information, see *VMRS* on page A8-658 and *VMSR* on page A8-660. For example:

```
VMRS <Rt>, FPSCR     ; Read  Floating-point System Control Register
VMSR FPSCR, <Rt>     ; Write Floating-point System Control Register
```

A2.6.5 VFPv3U

VFPv3 does not support the exception trap enable bits in the FPSCR, see *Floating-point Status and Control Register (FPSCR)* on page A2-28. All floating-point exceptions are untrapped.

The VFPv3U variant of the VFPv3 architecture implements the exception trap enable bits in the FPSCR, and provides exception handling as described in *VFP support code* on page B1-70. There is a separate trap enable bit for each of the six floating-point exceptions described in *Floating-point exceptions* on page A2-42. The VFPv3U architecture is otherwise identical to VFPv3.

Trapped exception handling never causes the corresponding cumulative exception bit of the FPSCR to be set to 1. If this behavior is desired, the trap handler routine must use a read, modify, write sequence on the FPSCR to set the cumulative exception bit.

VFPv3U is backwards compatible with VFPv2.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

FPSCR への書き込みは、プロセッサ動作のさまざまな側面に副作用を及ぼす可能性があります。これらの副作用は、すべて FPSCR への書き込みに同期します。つまり、これらの副作用は、実行ストリーム内でその命令よりも前にある命令からは可視でなく、その命令より後にある命令からは可視であることが保証されます。

FPSCR へのアクセス

FPSCR への読み出しまたは書き込みを行うには、`VMRS` および `VMSR` 命令を使用します。詳細については、*VMRS* P. A8-658 と *VMSR* P. A8-660 を参照して下さい。次に例を示します。

```
VMRS <Rt>, FPSCR     ; Read  Floating-point System Control Register
VMSR FPSCR, <Rt>     ; Write Floating-point System Control Register
```

A2.6.5 VFPv3U

VFPv3 では、FPSCR の例外トラップ イネーブルビットがサポートされていません。*浮動小数点ステータスおよび制御レジスタ (FPSCR)* P. A2-28 を参照して下さい。このため、浮動小数点例外は一切トラップされません。

VFPv3 アーキテクチャの VFPv3U バリアントは、FPSCR に例外トラップ イネーブルビットを実装し、*VFP サポートコード* P. B1-70 に説明されている例外処理を提供します。*浮動小数点例外* P. A2-42 に説明されている 6 つの浮動小数点例外のそれぞれについて、独立したトラップ イネーブルビットが存在します。それ以外の点では、VFPv3U は VFPv3 アーキテクチャと同じです。

トラップ例外の処理によって、その例外に対応する FPSCR の累積例外ビットが 1 にセットされることはありません。このビットをセットすることが必要な場合は、トラップハンドラ ルーチンが FPSCR に対して、読み出し－変更－書き込みシーケンスを使用し、累積例外ビットをセットする必要があります。

VFPv3U は VFPv2 と下位互換です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.7 Floating-point data types and arithmetic

The VFP extension supports single-precision (32-bit) and double-precision (64-bit) floating-point data types and arithmetic as defined by the IEEE 754 floating-point standard. It also supports the *ARM Standard modifications* to that arithmetic described in *Flush-to-zero* on page A2-39 and *NaN handling and the Default NaN* on page A2-41.

Trapped floating-point exception handling is supported in the VFPv3U variant only (see *VFPv3U* on page A2-31).

ARM standard floating-point arithmetic means IEEE 754 floating-point arithmetic with the ARM standard modifications and:

- the Round to Nearest rounding mode selected
- untrapped exception handling selected for all floating-point exceptions.

The Advanced SIMD extension only supports single-precision ARM standard floating-point arithmetic.

Note

Implementations of the VFP extension require *support code* to be installed in the system if trapped floating-point exception handling is required. See *VFP support code* on page B1-70.

They might also require support code to be installed in the system to support other aspects of their floating-point arithmetic. It is IMPLEMENTATION DEFINED which aspects of VFP floating-point arithmetic are supported in a system without support code installed.

Aspects of floating-point arithmetic that are implemented in support code are likely to run much more slowly than those that are executed in hardware.

ARM recommends that:

- To maximize the chance of getting high floating-point performance, software developers use ARM standard floating-point arithmetic.
- Software developers check whether their systems have support code installed, and if not, observe the IMPLEMENTATION DEFINED restrictions on what operations their VFP implementation can handle without support code.
- VFP implementation developers implement at least ARM standard floating-point arithmetic in hardware, so that it can be executed without any need for support code.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A2.7 浮動小数点データ型と算術演算

VFP 拡張機能は、IEEE 754 浮動小数点規格に定義されているとおり、単精度（32 ビット）および倍精度（64 ビット）の浮動小数点データ型と算術演算をサポートしています。また、算術演算への *ARM 標準変更*をサポートしています。これについては *Flush-to-Zero* P. A2-39 と *NaN の処理とデフォルト NaN* P. A2-41 に説明されています。

トラップ浮動小数点例外の処理は、VFPv3U バリアントでのみサポートされています（*VFPv3U* P. A2-31 参照）。

*ARM 標準浮動小数点算術演算*とは、IEEE 754 浮動小数点算術演算と ARM 標準変更を意味し、次の条件が適用されます。

- 近似値への丸め (RN) モードが選択されます。
- すべての浮動小数点例外について、トラップなしの例外処理が選択されます。

アドバンスト SIMD 拡張機能は、単精度の ARM 標準浮動小数点算術演算のみをサポートします。

注

トラップ浮動小数点例外の処理が必要な場合、VFP 拡張機能を実装する際に、*サポートコード*をシステムにインストールする必要があります。詳細については、*VFP サポートコード* P. B1-70 を参照して下さい。

浮動小数点算術演算の他の側面に対応するサポートコードをシステムにインストールすることが必要な場合もあります。サポートコードのインストールなしで、VFP 浮動小数点算術演算のどの側面がシステムでサポートされているかは、実装定義です。

サポートコードに実装される浮動小数点算術演算は、ハードウェアで実行されるより動作速度が大幅に遅くなる傾向にあります。

推奨される方針は次のとおりです。

- できるだけ大きな可能性で浮動小数点の高パフォーマンスを実現するため、ソフトウェア開発者は ARM 標準浮動小数点算術演算を使用して下さい。
- ソフトウェア開発者は、システムにサポートコードがインストールされているかどうかをチェックして下さい。インストールされていない場合、VFP 実装がサポートコードなしでどの演算を処理できるかに関する実装定義の制約に注意して下さい。
- VFP 実装開発者は、少なくとも ARM 標準浮動小数点算術演算をサポートコードなしで実行できるようハードウェア実装を行って下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.7.1 ARM standard floating-point input and output values

ARM standard floating-point arithmetic supports the following input formats defined by the IEEE 754 floating-point standard:

- Zeros.
- Normalized numbers.
- Denormalized numbers are flushed to 0 before floating-point operations. For details, see *Flush-to-zero* on page A2-39.
- NaNs.
- Infinities.

ARM standard floating-point arithmetic supports the Round to Nearest rounding mode defined by the IEEE 754 standard.

ARM standard floating-point arithmetic supports the following output result formats defined by the IEEE 754 standard:

- Zeros.
- Normalized numbers.
- Results that are less than the minimum normalized number are flushed to zero, see *Flush-to-zero* on page A2-39.
- NaNs produced in floating-point operations are always the default NaN, see *NaN handling and the Default NaN* on page A2-41.
- Infinities.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.7.1 ARM 標準浮動小数点の入力と出力の値

ARM 標準浮動小数点算術演算では、IEEE 754 浮動小数点規格に定義されている次の入力フォーマットがサポートされています。

- 0
- 正規化数
- 非正規化数は、浮動小数点演算の前に 0 にフラッシュされます。詳細については、*Flush-to-Zero* P. A2-39 を参照して下さい。
- NaN
- 無限大

ARM 標準浮動小数点算術演算では、IEEE 754 規格に定義されている、近似値への丸めモードがサポートされています。

ARM 標準浮動小数点算術演算では、IEEE 754 規格に定義されている、次の出力結果フォーマットがサポートされています。

- 0
- 正規化数
- 最小正規化数より小さい結果は、0 にフラッシュされます。*Flush-to-Zero* P. A2-39 を参照して下さい。
- 浮動小数点演算で生成される NaN は常にデフォルト NaN です。*NaN の処理とデフォルト NaN* P. A2-41 を参照して下さい。
- 無限大

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.7.2 Advanced SIMD and VFP single-precision format

The single-precision floating-point format used by the Advanced SIMD and VFP extensions is as defined by the IEEE 754 standard.

This description includes ARM-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs and signed zeros, see the IEEE 754 standard.

A single-precision value is a 32-bit word, and must be word-aligned when held in memory. It has the format:

31	30 … 23	22 … 0
S	exponent	fraction

The interpretation of the format depends on the value of the exponent field, bits [30:23]:

0 < exponent < `0xFF`

The value is a *normalized number* and is equal to:

$-1^S \times 2^{(\text{exponent} - 127)} \times (1.\text{fraction})$

The minimum positive normalized number is 2^{-126}, or approximately 1.175×10^{-38}.

The maximum positive normalized number is $(2 - 2^{-23}) \times 2^{127}$, or approximately 3.403×10^{38}.

exponent == 0

The value is either a zero or a *denormalized number*, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 when S==0

–0 when S==1.

These usually behave identically. In particular, the result is *equal* if +0 and –0 are compared as floating-point numbers. However, they yield different results in some circumstances. For example, the sign of the infinity produced as the result of dividing by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer comparison of the two words.

fraction != 0

The value is a denormalized number and is equal to:

$-1^S \times 2^{-126} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-149}, or approximately 1.401×10^{-45}.

Denormalized numbers are flushed to zero in the Advanced SIMD extension. They are optionally flushed to zero in the VFP extension. For details see *Flush-to-zero* on page A2-39.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A2.7.2 アドバンスト SIMD と VFP の単精度フォーマット

アドバンスト SIMD および VFP 拡張機能で使用される単精度浮動小数点フォーマットは、IEEE 754 規格に定義されているとおりです。

この説明には、規格では定義されていない ARM 固有の詳細が含まれています。これは、フォーマットとそれに含まれる値の概要を説明することを目的としています。完全な詳細、特に無限大、NaN、符号付き 0 の処理については、IEEE 754 規格を参照して下さい。

単精度値は 32 ビットワードで、メモリに保持されているときはワード境界でアラインしている必要があります。フォーマットは次のとおりです。

31	30 … 23	22 … 0
S	指数	仮数

フォーマットの解釈は、指数フィールドであるビット [30:23] の値によって決まります。

0 < 指数 < `0xFF`

値は*正規化数*で、次の式の結果で与えられます。

$-1^S \times 2^{(\text{指数} - 127)} \times (1.\text{仮数})$

正の最小正規化数は 2^{-126}、すなわち約 1.175×10^{-38} です。

正の最大正規化数は $(2 - 2^{-23}) \times 2^{127}$、すなわち約 3.403×10^{38} です。

指数 == 0

値は、仮数ビットによって決まり、0 または*非正規化数*です。

仮数 == 0

値は 0 です。0 には、明確に区別される 2 つの 0 があります。

+0 S == 0 の場合

-0 S == 1 の場合

通常、これらは同じように動作します。特に、+ 0 と - 0 を浮動小数点数として比較した場合、結果は*イコール*です。しかし、異なる結果を導く場合もあります。例えば、0 で除算した結果として生成される無限大の符合は、0 の符号によって決まります。2 つの 0 は、2 つのワードに対して整数比較を実行することによって区別できます。

仮数 != 0

値は非正規化数で、次の式で与えられます。

$-1^S \times 2^{-126} \times (0.\text{仮数})$

正の最小非正規化数は 2^{-149}、すなわち約 1.401×10^{-45} です。

アドバンスト SIMD 拡張機能では、非正規化数が 0 にフラッシュされます。VFP 拡張機能では、オプションで 0 にフラッシュされます。詳細については、*Flush-to-Zero* P. A2-39 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

exponent == `0xFF`

The value is either an *infinity* or a *Not a Number* (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+∞ When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-∞ When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

In the VFP architecture, the two types of NaN are distinguished on the basis of their most significant fraction bit, bit [22]:

bit [22] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit [22] == 1

The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

For details of the *default NaN* see *NaN handling and the Default NaN* on page A2-41.

Note

NaNs with different sign or fraction bits are distinct NaNs, but this does not mean you can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself. However, you can use integer comparisons to distinguish different NaNs.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

指数 == `0xFF`

値は仮数ビットによって決まり、*無限大*または*非数*(NaN) です。

仮数 == 0

値は無限大です。無限大には 2 種類あり、明確に区別されます。

+∞ S = 0 の場合。正規化数として正確に表現するには大きすぎるすべての正の数を表します。

−∞ S = 1 の場合。正規化数として正確に表現するには大きすぎる絶対値を持つすべての負の数を表します。

仮数 != 0

値は NaN で、*クワイエット NaN* または*シグナル NaN* のいずれかです。

VFP アーキテクチャでは、仮数の最上位ビットであるビット [22] により、2 つのタイプの NaN が区別されます。

ビット [22] = 0

NaN はシグナル NaN です。符号ビットはあらゆる値をとることができ、残りの仮数ビットもすべて 0 以外のあらゆる値をとることができます。

ビット [22] = 1

NaN はクワイエット NaN です。符号ビットと残りの仮数ビットはあらゆる値をとることができます。

デフォルト NaN の詳細については、*NaN の処理とデフォルト NaN* P. A2- 41 を参照して下さい。

注

符号ビットや仮数ビットが異なる NaN は区別される NaN ですが、浮動小数点比較命令を使用して区別できるという意味ではありません。これは、IEEE 754 規格では NaN について、それ自体も含め、何と比較しても*規則なし*と定められているためです。しかし、整数比較を使用して異なる NaN を区別することはできます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.7.3 VFP double-precision format

The double-precision floating-point format used by the VFP extension is as defined by the IEEE 754 standard.

This description includes VFP-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs and signed zeros, see the IEEE 754 standard.

A double-precision value consists of two 32-bit words, with the formats:

Most significant word:

31	30 … 20	19 … 0
S	exponent	fraction[51:32]

Least significant word:

31 … 0
fraction[31:0]

When held in memory, the two words must appear consecutively and must both be word-aligned. The order of the two words depends on the endianness of the memory system:

- In a little-endian memory system, the least significant word appears at the lower memory address and the most significant word at the higher memory address.
- In a big-endian memory system, the most significant word appears at the lower memory address and the least significant word at the higher memory address.

Double-precision values represent numbers, infinities and NaNs in a similar way to single-precision values, with the interpretation of the format depending on the value of the exponent:

0 < exponent < `0x7FF`

The value is a normalized number and is equal to:

$-1^{S} \times 2^{exponent-1023} \times (1.fraction)$

The minimum positive normalized number is 2^{-1022}, or approximately 2.225×10^{-308}.

The maximum positive normalized number is $(2 - 2^{-52}) \times 2^{1023}$, or approximately 1.798×10^{308}.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros that behave analogously to the two single-precision zeros:

+0 when S==0

–0 when S==1.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A2.7.3 VFP 倍精度フォーマット

VFP 拡張機能で使用される倍精度の浮動小数点フォーマットは、IEEE 754 規格に定義されているとおりです。

この説明には、規格では定義されていない VFP 固有の詳細が含まれています。これは、フォーマットとそれに含まれる値の概要を説明することを目的としています。完全な詳細、特に無限大、NaN、符号付き 0 の処理については、IEEE 754 規格を参照して下さい。

倍精度の値は 2 つの 32 ビットワードで構成され、次のフォーマットで表現されます。

最上位ワード

31	30 … 20	19 … 0
S	指数	仮数 [51:32]

最下位ワード

31 … 0
仮数 [31:0]

メモリに格納する場合、2 つのワードは連続し、どちらもワード境界でアラインしている必要があります。2 つのワードの順序は、メモリシステムのエンディアン形式によって決まります。

- リトルエンディアンのメモリシステムでは、最下位ワードは下位のメモリアドレス、最上位ワードは上位のメモリアドレスに置かれます。
- ビッグエンディアンのメモリシステムでは、最上位ワードは下位のメモリアドレス、最下位ワードは上位のメモリアドレスに置かれます。

倍精度の値は、単精度の値と同様の形式で数値、無限大、NaN を表現します。フォーマットの解釈は指数の値によって決まります。

0 < 指数 < `0x7FF`

値は正規化数で、次の式で与えられます。

$-1^{S} \times 2^{指数-1023} \times (1.仮数)$

正の最小正規化数は 2^{-1022}、すなわち約 2.225×10^{-308} です。

正の最大正規化数は $(2 - 2^{-52}) \times 2^{1023}$、すなわち約 1.798×10^{308} です。

指数 == 0

値は仮数ビットによって決まり、0 または非正規化数です。

仮数 == 0

値は 0 です。0 には 2 種類あり、単精度の 2 つの 0 と同様に動作します。

+0 S == 0 の場合

-0 S == 1 の場合

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

fraction != 0

The value is a denormalized number and is equal to:

$1^{-S} \times 2^{-1022} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-1074}, or approximately 4.941×10^{-324}.

Optionally, denormalized numbers are flushed to zero in the VFP extension. For details see *Flush-to-zero* on page A2-39.

exponent == `0x7FF`

The value is either an infinity or a NaN, depending on the fraction bits:

fraction == 0

the value is an infinity. As for single-precision, there are two infinities:

+∞ Plus infinity, when S==0

-∞ Minus infinity, when S==1.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

In the VFP architecture, the two types of NaN are distinguished on the basis of their most significant fraction bit, bit [19] of the most significant word:

bit [19] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit [19] == 1

The NaN is a quiet NaN. The sign bit and the remaining fraction bits can take any value.

For details of the *default NaN* see *NaN handling and the Default NaN* on page A2-41.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

仮数 != 0

値は非正規化数で、次の式で与えられます。

$1^{-S} \times 2^{-1022} \times (0.\text{仮数})$

正の最小非正規化数は 2^{-1074}、すなわち約 4.941×10^{-324} です。

VFP 拡張機能では、非正規化数がオプションで 0 にフラッシュされます。詳細については、*Flush-to-Zero* P. A2-39 を参照して下さい。

指数 == `0x7FF`

値は仮数ビットによって決まり、無限大または NaN です。

仮数 == 0

値は無限大です。単精度と同様に、無限大には 2 種類あります。

+∞ S = 0 の場合、プラスの無限大

−∞ S = 1 の場合、マイナスの無限大

仮数 != 0

値は NaN で、*クワイエット NaN* または*シグナル NaN* のいずれかです。

VFP アーキテクチャでは、仮数の最上位ビット、つまり最上位ワードのビット [19] により、2 つのタイプの NaN が区別されます。

ビット [19] = 0

NaN はシグナル NaN です。符号ビットはあらゆる値をとることができ、残りの仮数ビットもすべて 0 以外のあらゆる値をとることができます。

ビット [19] = 1

NaN はクワイエット NaN です。符号ビットと残りの仮数ビットはあらゆる値をとることができます。

デフォルト NaN の詳細については、*NaN の処理とデフォルト NaN* P. A2-41 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.7.4 Advanced SIMD and VFP half-precision formats

Two half-precision floating-point formats are used by the half-precision extensions to Advanced SIMD and VFP:

- IEEE half-precision, as described in the revised IEEE 754 standard
- Alternative half-precision.

The description of IEEE half-precision includes ARM-specific details that are left open by the standard, and is only an introduction to the formats and to the values they can contain. For more information, especially on the handling of infinities, NaNs and signed zeros, see the IEEE 754 standard.

For both half-precision floating-point formats, the layout of the 16-bit number is the same. The format is:

The interpretation of the format depends on the value of the exponent field, bits[14:10] and on which half-precision format is being used.

0 < exponent < `0x1F`

The value is a normalized number and is equal to:

$-1^S \times 2^{((exponent-15)} \times (1.fraction)$

The minimum positive normalized number is 2^{-14}, or approximately 6.104×10^{-5}.

The maximum positive normalized number is $(2 - 2^{-10}) \times 2^{15}$, or 65504.

Larger normalized numbers can be expressed using the alternative format when the exponent == `0x1F`.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 when S==0

–0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$-1^S \times 2^{-14} \times (0.fraction)$

The minimum positive denormalized number is 2^{-25}, or approximately 2.980×10^{-8}.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.7.4 アドバンスト SIMD と VFP の半精度フォーマット

アドバンスト SIMD および VFP の半精度拡張機能では、2 つの半精度浮動小数点フォーマットが使用されます。

- 改訂された IEEE 754 規格に記述されている IEEE 半精度
- 代替半精度

IEEE 半精度の説明には、規格には定められていない ARM 固有の詳細も含まれていますが、ここで説明するのは各フォーマットと、それらに含めることができる値の概要のみです。詳細、特に無限大、NaN、符号付き 0 の処理については、IEEE 754 規格を参照して下さい。

どちらの半精度浮動小数点フォーマットでも、16 ビット数値のレイアウトは同じです。フォーマットは次のとおりです。

フォーマットの解釈は、指数フィールドであるビット [14:10] の値と、使用する半精度フォーマットによって決まります。

0 < 指数 < `0x1F`

値は正規化数で、次の式で与えられます。

$-1^S \times 2^{(指数-15)} \times (1.仮数)$

正の最小正規化数は 2^{-14}、すなわち約 6.104×10^{-5} です。

正の最大正規化数は $(2 - 2^{-10}) \times 2^{15}$、すなわち 65504 です。

これより大きな正規化数は、指数 == `0x1F` のとき、代替フォーマットを使用して表現できます。

指数 == 0

値は仮数ビットによって決まり、0 または非正規化数です。

仮数 == 0

値は 0 です。0 には、明確に区別される 2 つの 0 があります。

+0 S == 0 の場合

-0 S == 1 の場合

仮数 != 0

値は非正規化数で、次の式で与えられます。

$-1^S \times 2^{-14} \times (0.仮数)$

正の最小非正規化数は 2^{-25}、すなわち約 2.980×10^{-8} です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

exponent == 0x1F

The value depends on which half-precision format is being used:

IEEE Half-precision

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+∞	When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.
-∞	When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a quiet NaN or a signaling NaN. The two types of NaN are distinguished by their most significant fraction bit, bit [9]:

bit [9] == 0	The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.
bit [9] == 1	The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

Alternative Half-precision

The value is a normalized number and is equal to:

$-1^S \times 2^{16} \times (1.\text{fraction})$

The maximum positive normalized number is $(2-2^{-10}) \times 2^{16}$ or 131008.

A2.7.5 Flush-to-zero

The performance of floating-point implementations can be significantly reduced when performing calculations involving denormalized numbers and Underflow exceptions. In particular this occurs for implementations that only handle normalized numbers and zeros in hardware, and invoke support code to handle any other types of value. For an algorithm where a significant number of the operands and intermediate results are denormalized numbers, this can result in a considerable loss of performance.

In many of these algorithms, this performance can be recovered, without significantly affecting the accuracy of the final result, by replacing the denormalized operands and intermediate results with zeros. To permit this optimization, VFP implementations have a special processing mode called *Flush-to-zero* mode. Advanced SIMD implementations always use Flush-to-zero mode.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

指数 == 0x1F

値は、使用する半精度フォーマットによって決まります。

IEEE 半精度

値は仮数ビットによって決まり、無限大または非数 (NaN) です。

仮数 == 0

値は無限大です。無限大には 2 種類あり、明確に区別されます。

+∞	S == 0 の場合。正規化数として正確に表現するには大きすぎるすべての正の数を表します。
−∞	S == 1 の場合。正規化数として正確に表現するには大きすぎる絶対値を持つすべての負の数を表します。

仮数 != 0

値は NaN で、クワイエット NaN またはシグナル NaN のいずれかです。2 つのタイプの NaN は、仮数の最上位ビットであるビット [9] によって区別されます。

ビット [9] == 0	NaN はシグナル NaN です。符号ビットはあらゆる値をとることができ、残りの仮数ビットもすべて 0 以外のあらゆる値をとることができます。
ビット [9] == 1	NaN はクワイエット NaN です。符号ビットと残りの仮数ビットはあらゆる値をとることができます。

代替半精度

値は正規化数で、次の式で与えられます。

$-1^S \times 2^{16} \times (1.\text{仮数})$

正の最大正規化数は $(2 - 2^{-10}) \times 2^{16}$、すなわち 131008 です。

A2.7.5 Flush-to-Zero

浮動小数点実装のパフォーマンスは、非正規化数とアンダーフロー例外に関わる計算を実行する場合、大幅に低下することがあります。これは特に、正規化数と 0 だけをハードウェアで処理し、他のタイプの値はサポートコードを呼び出して処理する実装に当てはまります。オペランドおよび中間結果に多くの非正規化数が存在するアルゴリズムでは、この理由からパフォーマンスが大幅に低下する可能性があります。

このようなアルゴリズムの多くでは、非正規化のオペランドと中間結果を 0 に置き換えると、最終結果の精度に大きく影響を与えることなく、パフォーマンスを回復できます。この最適化を許可するため、VFP 実装には *Flush-to-zero* モードと呼ばれる特殊な処理モードがあります。アドバンスト SIMD 実装では、常に Flush-to-Zero モードが使用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Behavior in Flush-to-zero mode differs from normal IEEE 754 arithmetic in the following ways:

- All inputs to floating-point operations that are double-precision de-normalized numbers or single-precision de-normalized numbers are treated as though they were zero. This causes an Input Denormal exception, but does not cause an Inexact exception. The Input Denormal exception occurs only in Flush-to-zero mode.

  The FPSCR contains a cumulative exception bit FPSCR.IDC and trap enable bit FPSCR.IDE corresponding to the Input Denormal exception. For details of how these are used when processing the exception see *Advanced SIMD and VFP system registers* on page A2-28.

  The occurrence of all exceptions except Input Denormal is determined using the input values after flush-to-zero processing has occurred.

- The result of a floating-point operation is flushed to zero if the result of the operation before rounding satisfies the condition:

  $0 < \texttt{Abs(result)} < \text{MinNorm}$, where:

  — $\text{MinNorm} == 2^{-126}$ for single-precision

  — $\text{MinNorm} == 2^{-1022}$ for double-precision.

  This causes the FPSCR.UFC bit to be set to 1, and prevents any Inexact exception from occurring for the operation.

  Underflow exceptions occur only when a result is flushed to zero.

  In a VFPv2 or VFPv3U implementation Underflow exceptions that occur in Flush-to-zero mode are always treated as untrapped, even when the Underflow trap enable bit, FPSCR.UFE, is set to 1.

- An Inexact exception does not occur if the result is flushed to zero, even though the final result of zero is not equivalent to the value that would be produced if the operation were performed with unbounded precision and exponent range.

For information on the FPSCR bits see *Floating-point Status and Control Register (FPSCR)* on page A2-28.

When an input or a result is flushed to zero the value of the sign bit of the zero is determined as follows:

- In VFPv3 or VFPv3U, it is preserved. That is, the sign bit of the zero matches the sign bit of the input or result that is being flushed to zero.

- In VFPv2, it is IMPLEMENTATION DEFINED whether it is preserved or always positive. The same choice must be made for all cases of flushing an input or result to zero.

Flush-to-zero mode has no effect on half-precision numbers that are inputs to floating-point operations, or results from floating-point operations.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

Flush-to-zero モードの動作は、通常の IEEE 754 算術演算とは次の点が異なります。

- 浮動小数点演算への倍精度非正規化数または単精度非正規化数の入力はすべて、0 として扱われます。これにより入力非正規化例外が発生しますが、不正確例外は発生しません。入力非正規化例外は、Flush-to-zero モードでのみ発生します。

  FPSCR には、入力非正規化例外に対応する累積例外ビット FPSCR.IDC と、トラップイネーブルビット FPSCR.IDE が含まれています。例外処理におけるこれらの詳しい用法については、*アドバンスト SIMD と VFP のシステムレジスタ* P. A2-28 を参照して下さい。

  入力非正規化を除くすべての例外の発生は、Flush-to-Zero 処理が発生した後で、入力値を使用して決定されます。

- 浮動小数点演算の結果が丸め前に次の条件を満たしている場合、その結果は 0 にフラッシュされます。

  $0 < \texttt{Abs(result)} < \text{MinNorm}$。MinNorm は次の値です。

  – 単精度の場合、$\text{MinNorm} == 2^{-126}$

  – 倍精度の場合、$\text{MinNorm} == 2^{-1022}$

  これにより FPSCR.UFC ビットは 1 にセットされ、不正確例外が発生することを防ぎます。

  結果が 0 にフラッシュされた場合のみ、アンダーフロー例外が発生します。

  VFPv2 または VFPv3U 実装では、Flush-to-Zero モードで発生するアンダーフロー例外は、アンダーフロートラップ イネーブルビット FPSCR.UFE が 1 にセットされている場合でも、常にトラップなしとして扱われます。

- 結果が 0 にフラッシュされた場合、0 という最終結果が、精度と指数範囲の制約なしで演算を実行したときに生成される値と等しくなくても、不正確例外は発生しません。

FPSCR ビットの詳細については、*浮動小数点ステータスおよび制御レジスタ (FPSCR)* P. A2-28 を参照して下さい。

入力または結果が 0 にフラッシュされた場合、0 の符号ビットの値は次のように決定されます。

- VFPv3 または VFPv3U では、保存されます。つまり、0 の符号ビットは、0 にフラッシュされた入力や結果の符号ビットと一致します。

- VFPv2 では、保存されるか常に正かは実装定義です。入力や結果を 0 にフラッシュするすべての場合について、同じ選択が行われる必要があります。

Flush-to-Zero モードは、浮動小数点演算への入力または浮動小数点演算の結果としての半精度数には影響を与えません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

Flush-to-zero mode is incompatible with the IEEE 754 standard, and must not be used when IEEE 754 compatibility is a requirement. Flush-to-zero mode must be treated with care. Although it can lead to a major performance increase on many algorithms, there are significant limitations on its use. These are application dependent:

- On many algorithms, it has no noticeable effect, because the algorithm does not normally use denormalized numbers.
- On other algorithms, it can cause exceptions to occur or seriously reduce the accuracy of the results of the algorithm.

A2.7.6 NaN handling and the Default NaN

The IEEE 754 standard specifies that:

- an operation that produces an Invalid Operation floating-point exception generates a quiet NaN as its result if that exception is untrapped
- an operation involving a quiet NaN operand, but not a signaling NaN operand, returns an input NaN as its result.

The VFP behavior when Default NaN mode is disabled adheres to this with the following extra details, where the *first operand* means the first argument to the pseudocode function call that describes the operation:

- If an untrapped Invalid Operation floating-point exception is produced because one of the operands is a signaling NaN, the quiet NaN result is equal to the signaling NaN with its most significant fraction bit changed to 1. If both operands are signaling NaNs, the result is produced in this way from the first operand.
- If an untrapped Invalid Operation floating-point exception is produced for other reasons, the quiet NaN result is the Default NaN.
- If both operands are quiet NaNs, the result is the first operand.

The VFP behavior when Default NaN mode is enabled, and the Advanced SIMD behavior in all circumstances, is that the Default NaN is the result of all floating-point operations that:

- generate untrapped Invalid Operation floating-point exceptions
- have one or more quiet NaN inputs.

Table A2-6 on page A2-42 shows the format of the default NaN for ARM floating-point processors.

Default NaN mode is selected for VFP by setting the FPSCR.DN bit to 1, see *Floating-point Status and Control Register (FPSCR)* on page A2-28.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

Flush-to-Zero モードは、IEEE 754 規格と互換性を持たないため、IEEE 754 規格との互換性が必要な場合には使用できません。Flush-to-Zero モードは注意して扱う必要があります。多くのアルゴリズムにおいてパフォーマンスを大幅に高める可能性がありますが、用法にはかなりの制限があります。どのような制限があるかはアプリケーションに依存します。

- 多くのアルゴリズムでは、通常は非正規化数を使用しないため、これらのアルゴリズムにおいては顕著な影響はありません。
- 他のアルゴリズムでは、例外が発生したり、アルゴリズムの結果の精度が大きく低下したりする可能性があります。

A2.7.6 NaN の処理とデフォルト NaN

IEEE 754 規格には、次の条件が指定されています。

- 無効演算浮動小数点例外を生成する演算は、その例外がトラップされない場合、結果としてクワイエット NaN を生成する。
- クワイエット NaN オペランドを含み、シグナル NaN オペランドを含まない演算は、入力された NaN を結果として返す。

デフォルト NaN モードが不可能な場合の VFP 動作はこれに加え、次のような詳細に従います。*最初のオペランド*とは、演算を記述する擬似コード関数呼び出しへの最初の引数を意味します。

- オペランドの 1 つがシグナル NaN であることが原因で、トラップされない無効演算浮動小数点例外が発生すると、クワイエット NaN の結果はシグナル NaN の結果と等しくなり、最上位の仮数ビットが 1 に変更されます。両方のオペランドがシグナル NaN の場合、結果は最初のオペランドからこの方法で生成されます。
- トラップされない無効演算浮動小数点例外が他の理由で発生した場合、クワイエット NaN の結果はデフォルト NaN です。
- 両方のオペランドがクワイエット NaN の場合、結果は最初のオペランドです。

デフォルト NaN モードが可能である場合の VFP の動作、およびすべての場合におけるアドバンスト SIMD の動作では、次のすべての浮動小数点演算において、結果はデフォルト NaN です。

- トラップされない無効演算浮動小数点例外を生じる浮動小数点演算
- 1 つ以上のクワイエット NaN 入力を持つ浮動小数点演算

ARM 浮動小数点プロセッサのデフォルト NaN のフォーマットを、P. A2-42 の表 A2-6 に示します。

VFP のデフォルト NaN モードは、FPSCR.DN ビットを 1 にセットすることによって選択されます。*浮動小数点ステータスおよび制御レジスタ (FPSCR)* P. A2-28 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Other aspects of the functionality of the Invalid Operation exception are not affected by Default NaN mode. These are that:

- If untrapped, it causes the FPSCR.IOC bit be set to 1.
- If trapped, it causes a user trap handler to be invoked. This is only possible in VFPv2 and VFPv3U.

Table A2-6 Default NaN encoding

	Half-precision, IEEE Format	Single-precision	Double-precision
Sign bit	0	0[a]	0[a]
Exponent	`0x1F`	`0xFF`	`0x7FF`
Fraction	Bit[9] == 1, bits[8:0] == 0	bit [22] == 1, bits [21:0] == 0	bit [51] == 1, bits [50:0] == 0

a. In VFPv2, the sign bit of the Default NaN is UNKNOWN.

A2.7.7 Floating-point exceptions

The Advanced SIMD and VFP extensions record the following floating-point exceptions in the FPSCR cumulative flags, see *Floating-point Status and Control Register (FPSCR)* on page A2-28:

IOC Invalid Operation. The flag is set to 1 if the result of an operation has no mathematical value or cannot be represented. Cases include infinity * 0, +infinity + (–infinity), for example. These tests are made after flush-to-zero processing. For example, if flush-to-zero mode is selected, multiplying a denormalized number and an infinity is treated as 0 * infinity and causes an Invalid Operation floating-point exception.

IOC is also set on any floating-point operation with one or more signaling NaNs as operands, except for negation and absolute value, as described in *Negation and absolute value* on page A2-47.

DZC Division by Zero. The flag is set to 1 if a divide operation has a zero divisor and a dividend that is not zero, an infinity or a NaN. These tests are made after flush-to-zero processing, so if flush-to-zero processing is selected, a denormalized dividend is treated as zero and prevents Division by Zero from occurring, and a denormalized divisor is treated as zero and causes Division by Zero to occur if the dividend is a normalized number.

For the reciprocal and reciprocal square root estimate functions the dividend is assumed to be +1.0. This means that a zero or denormalized operand to these functions sets the DZC flag.

OFC Overflow. The flag is set to 1 if the absolute value of the result of an operation, produced after rounding, is greater than the maximum positive normalized number for the destination precision.

UFC Underflow. The flag is set to 1 if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, and the rounded result is inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

無効演算例外の機能の他の側面は、デフォルト NaN モードによって影響を受けません。具体的には次のとおりです。

- トラップされない場合、FPSCR.IOC ビットを 1 にセットします。
- トラップされる場合、ユーザトラップ ハンドラを呼び出します。これは、VFPv2 と VFPv3U でのみ可能です。

表 A2-6 デフォルト NaN のエンコード

	半精度、IEEE フォーマット	単精度	倍精度
符号ビット	0	0[a]	0[a]
指数	`0x1F`	`0xFF`	`0x7FF`
仮数	ビット [9] == 1、 ビット [8:0] == 0	ビット [22] == 1、 ビット [21:0] == 0	ビット [51] == 1、 ビット [50:0] == 0

a. VFPv2 では、デフォルト NaN の符号ビットが未知です。

A2.7.7 浮動小数点例外

アドバンスト SIMD および VFP 拡張機能は、次の浮動小数点例外を FPSCR 累積フラグに記録します。*浮動小数点ステータスおよび制御レジスタ (FPSCR)* P. A2-28 を参照して下さい。

IOC 無効演算。演算の結果が数学的な値を持たない場合や表現できない場合、フラグが 1 にセットされます。例えば、無限大 * 0、+ 無限大 + (- 無限大) などです。これらのテストは、Flush-to-Zero 処理の後で行われます。例えば、Flush-to-Zero モードが選択されている場合、非正規化数と無限大との乗算は 0 * 無限大として扱われ、無効演算浮動小数点例外を引き起こします。

IOC は、浮動小数点演算でオペランドの 1 つ以上がシグナル NaN である場合にもセットされます。ただし、*否定と絶対値* P. A2-47 に説明されているように否定と絶対値は除きます。

DZC 0 による除算。除算の除数が 0 で、被除数が 0、無限大、NaN でない場合、フラグが 1 にセットされます。これらのテストは、Flush-to-Zero 処理の後で行われます。このため Flush-to-Zero 処理が選択されている場合、非正規化数の被除数は 0 として扱われるため、0 による除算は発生しません。また、非正規化数の除数が 0 として扱われるため、被除数が正規化数の場合に 0 による除算が発生します。

逆数および逆数平方根概算関数の場合、被除数は + 1.0 と仮定されます。つまり、これらの関数に 0 または被正規化数のオペランドが指定された場合、DZC フラグがセットされます。

OFC オーバフロー。演算結果の丸め後の絶対値が、デスティネーションの精度で表現可能な最大の正の正規化数を超えている場合、フラグが 1 にセットされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The criteria for the Underflow exception to occur are different in Flush-to-zero mode. For details, see *Flush-to-zero* on page A2-39.

IXC Inexact. The flag is set to 1 if the result of an operation is not equivalent to the value that would be produced if the operation were performed with unbounded precision and exponent range.

The criteria for the Inexact exception to occur are different in Flush-to-zero mode. For details, see *Flush-to-zero* on page A2-39.

IDC Input Denormal. The flag is set to 1 if a denormalized input operand is replaced in the computation by a zero, as described in *Flush-to-zero* on page A2-39.

With the Advanced SIMD extension and the VFPv3 extension these are non-trapping exceptions and the data-processing instructions do not generate any trapped exceptions.

With the VFPv2 and VFPv3U extensions:

- These exceptions can be trapped, by setting trap enable flags in the FPSCR, see *VFPv3U* on page A2-31. Trapped floating-point exceptions are delivered to user code in an IMPLEMENTATION DEFINED fashion.
- The definitions of the floating-point exceptions change as follows:
  - — if the Underflow exception is trapped, it occurs if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, regardless of whether the rounded result is inexact
  - — higher priority trapped exceptions can prevent lower priority exceptions from occurring, as described in *Combinations of exceptions* on page A2-44.

Table A2-7 shows the default results of the floating-point exceptions:

Table A2-7 Floating-point exception default results

Exception type	Default result for positive sign	Default result for negative sign
IOC, Invalid Operation	Quiet NaN	Quiet NaN
DZC, Division by Zero	+∞ (plus infinity)	−∞ (minus infinity)
OFC, Overflow	RN, RP: +∞ (plus infinity) RM, RZ: +MaxNorm	RN, RM: −∞ (minus infinity) RP, RZ: –MaxNorm
UFC, Underflow	Normal rounded result	Normal rounded result
IXC, Inexact	Normal rounded result	Normal rounded result
IDC, Input Denormal	Normal rounded result	Normal rounded result

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

UFC アンダーフロー。演算結果の丸め前の絶対値が、デスティネーションの精度で表現可能な最小の正の正規化数より小さく、丸め後の結果が不正確な場合、フラグが 1 にセットされます。

Flush-to-Zero モードでは、アンダーフロー例外の発生する基準が異なります。詳細については、*Flush-to-Zero* P. A2-39 を参照して下さい。

IXC 不正確。演算の結果が、精度と指数範囲の制約なしで演算を実行したときに生成される値と等しくない場合、フラグが 1 にセットされます。

Flush-to-Zero モードでは、不正確例外の発生する基準が異なります。詳細については、*Flush-to-Zero* P. A2-39 を参照して下さい。

IDC 入力非正規化。非正規化入力オペランドが計算で 0 に置き換えられた場合、フラグが 1 にセットされます。*Flush-to-Zero* P. A2-39 の説明を参照して下さい。

アドバンスト SIMD 拡張機能と VFPv3 拡張機能では、これらはトラップなしの例外であり、データ処理命令でトラップ例外は生成されません。

VFPv2 および VFPv3U 拡張機能では、次の条件が適用されます。

- FPSCR のトラップイネーブル フラグをセットすることにより、これらの例外をトラップできます。*VFPv3U* P. A2-31 を参照して下さい。トラップされた浮動小数点例外がユーザコードへ提供される方法は、実装定義です。
- 浮動小数点例外の定義が、次のように変わります。
  - – アンダーフロー例外がトラップされる場合、その例外は、演算結果の丸め前の絶対値が、デスティネーションの精度で表現可能な最小の正の正規化数より小さいとき、丸め後の結果が不正確かどうかに関係なく発生します。
  - – 優先度の高いトラップ例外によって、優先度の低い例外の発生の発生が抑制される場合があります。*例外の組み合わせ* P. A2-44 を参照して下さい。

浮動小数点例外のデフォルトの結果を、表 A2-7 に示します。

表 A2-7 浮動小数点例外のデフォルトの結果

例外のタイプ	正の符号のデフォルト結果	負の符号のデフォルト結果
IOC、無効演算	クワイエット NaN	クワイエット NaN
DZC、0 による除算	+ ∞ （プラス無限大）	- ∞ （マイナス無限大）
OFC、オーバフロー	RN、RP: + ∞ （プラス無限大） RM、RZ: + MaxNorm	RN、RM: - ∞ （マイナス無限大） RP、RZ: - MaxNorm
UFC、アンダーフロー	（通常の丸め結果）	（通常の丸め結果）
IXC、不正確	（通常の丸め結果）	（通常の丸め結果）
IDC、入力非正規化	（通常の丸め結果）	（通常の丸め結果）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In Table A2-7 on page A2-43:

MaxNorm The maximum normalized number of the destination precision

RM Round towards Minus Infinity mode, as defined in the IEEE 754 standard

RN Round to Nearest mode, as defined in the IEEE 754 standard

RP Round towards Plus Infinity mode, as defined in the IEEE 754 standard

RZ Round towards Zero mode, as defined in the IEEE 754 standard

- For Invalid Operation exceptions, for details of which quiet NaN is produced as the default result see *NaN handling and the Default NaN* on page A2-41.
- For Division by Zero exceptions, the sign bit of the default result is determined normally for a division. This means it is the exclusive OR of the sign bits of the two operands.
- For Overflow exceptions, the sign bit of the default result is determined normally for the overflowing operation.

Combinations of exceptions

The following pseudocode functions perform *floating-point operations*:

```
FixedToFP()
FPAbs()
FPAdd()
FPCompare()
FPCompareGE()
FPCompareGT()
FPDiv()
FPDoubleToSingle()
FPMax()
FPMin()
FPMul()
FPNeg()
FPRecipEstimate()
FPRecipStep()
FPRSqrtEstimate()
FPRSqrtStep()
FPSingleToDouble()
FPSqrt()
FPSub()
FPToFixed()
```

All of these operations except `FPAbs()` and `FPNeg()` can generate floating-point exceptions.

More than one exception can occur on the same operation. The only combinations of exceptions that can occur are:

- Overflow with Inexact
- Underflow with Inexact
- Input Denormal with other exceptions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

P. A2-43 の表 A2-7 の記号の意味は次のとおりです。

MaxNorm デスティネーション精度で表現可能な最大の正規化数

RM IEEE 754 規格で定義されている、マイナス無限大への丸めモード

RN IEEE 754 規格で定義されている、近似値への丸めモード

RP IEEE 754 規格で定義されている、プラス無限大への丸めモード

RZ IEEE 754 規格で定義されている、0 への丸めモード

- 無効演算例外において、どのクワイエット NaN がデフォルト結果として生成されるかの詳細については、*NaN の処理とデフォルト NaN* P. A2-41 を参照して下さい。
- 0 による除算例外において、デフォルト結果の符合ビットは除算の通常どおり決定されます。つまり、2 つのオペランドの符号ビットの排他的論理和です。
- オーバフロー例外において、デフォルト結果の符合ビットはオーバフロー演算の通常どおり決定されます。

例外の組み合わせ

次の擬似コード関数は、*浮動小数点演算*を実行します。

```
FixedToFP()
FPAbs()
FPAdd()
FPCompare()
FPCompareGE()
FPCompareGT()
FPDiv()
FPDoubleToSingle()
FPMax()
FPMin()
FPMul()
FPNeg()
FPRecipEstimate()
FPRecipStep()
FPRSqrtEstimate()
FPRSqrtStep()
FPSingleToDouble()
FPSqrt()
FPSub()
FPToFixed()
```

`FPAbs()` と `FPNeg()` を除く演算はすべて、浮動小数点例外を生成することがあります。

同じ演算で 2 つ以上の例外が発生することもあります。発生し得る例外の組み合わせは次のものだけです。

- オーバフローと不正確
- アンダーフローと不正確
- 入力非正規化とその他の例外

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When none of the exceptions caused by an operation are trapped, any exception that occurs causes the associated cumulative flag in the FPSCR to be set.

When one or more exceptions caused by an operation are trapped, the behavior of the instruction depends on the priority of the exceptions. The Inexact exception is treated as lowest priority, and Input Denormal as highest priority:

- If the higher priority exception is trapped, its trap handler is called. It is IMPLEMENTATION DEFINED whether the parameters to the trap handler include information about the lower priority exception. Apart from this, the lower priority exception is ignored in this case.
- If the higher priority exception is untrapped, its cumulative bit is set to 1 and its default result is evaluated. Then the lower priority exception is handled normally, using this default result.

Some floating-point instructions specify more than one floating-point operation, as indicated by the pseudocode descriptions of the instruction. In such cases, an exception on one operation is treated as higher priority than an exception on another operation if the occurrence of the second exception depends on the result of the first operation. Otherwise, it is UNPREDICTABLE which exception is treated as higher priority.

For example, a `VMLA.F32` instruction specifies a floating-point multiplication followed by a floating-point addition. The addition can generate Overflow, Underflow and Inexact exceptions, all of which depend on both operands to the addition and so are treated as lower priority than any exception on the multiplication. The same applies to Invalid Operation exceptions on the addition caused by adding opposite-signed infinities.

The addition can also generate an Input Denormal exception, caused by the addend being a denormalized number while in Flush-to-zero mode. It is UNPREDICTABLE which of an Input Denormal exception on the addition and an exception on the multiplication is treated as higher priority, because the occurrence of the Input Denormal exception does not depend on the result of the multiplication. The same applies to an Invalid Operation exception on the addition caused by the addend being a signaling NaN.

Note

Like other details of VFP instruction execution, these rules about exception handling apply to the overall results produced by an instruction when the system uses a combination of hardware and support code to implement it. See *VFP support code* on page B1-70 for more information.

These principles also apply to the multiple floating-point operations generated by VFP instructions in the deprecated VFP vector mode of operation. For details of this mode of operation see Appendix F *VFP Vector Operation Support*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

演算によって発生する例外のどれもトラップされない場合、発生する例外はすべて、FPSCR にある対応の累積フラグをセットします。

演算によって発生する例外の 1 つ以上がトラップされる場合、命令の動作はその例外の優先度によって異なります。不正確例外が最も優先度の低い例外、入力非正規化例外が最も優先度の高い例外として扱われます。

- 優先度の高い例外がトラップされる場合、トラップハンドラが呼び出されます。トラップハンドラへのパラメータに優先度の低い例外に関する情報が含まれるかどうかは、実装定義です。それ以外の点では、優先度の低い例外は無視されます。
- 優先度の高い例外がトラップされない場合、累積ビットは 1 にセットされ、デフォルトの結果が評価されます。その後で、優先度の低い例外が、このデフォルト結果を使用して通常に処理されます。

一部の浮動小数点命令は、命令の擬似コードの説明に示されているように、2 つ以上の浮動小数点演算を指定しています。このような場合、1 つの演算の結果によって 2 番目の演算で例外が発生するかどうかの違いが生じる場合は、最初の演算で発生する例外は、後の演算で発生する例外よりも高い優先度として扱われます。そうでない場合、どちらの例外が高い優先度で扱われるかは予測不能です。

例えば、`VMLA.F32` 命令は、浮動小数点乗算の後に浮動小数点加算を指定しています。加算ではオーバフロー、アンダーフロー、不正確例外が生成される可能性がありますが、すべて加算への両方のオペランドに依存するため、乗算における例外よりも低い優先度として扱われます。符号の異なる無限大の加算によって発生する加算の無効演算例外についても同じ条件が適用されます。

加算は、Flush-to-Zero モードで加数が非正規化数の場合、入力非正規化例外を生成することもあります。入力非正規化例外の発生は乗算の結果に依存しないため、加算における入力非正規化例外と乗算における例外のどちらが高い優先度として扱われるかは予測不能です。加数がシグナル NaN であることによって発生する加算の無効演算例外についても同じ条件が適用されます。

注

VFP 命令実行の他の詳細と同様に、例外処理に関するこれらの規則も、システムがハードウェアとサポートコードの組み合わせを使用して実装した命令が生成する結果全体に該当します。詳細については、*VFP サポートコード* P. B1-70 を参照して下さい。

これらの原則は、推奨されていない VFP ベクタモードでの動作で VFP 命令が生成する複数の浮動小数点演算にも該当します。この非推奨モードの詳細については、付録 F *「VFP ベクタ演算のサポート」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.7.8 Pseudocode details of floating-point operations

This section contains pseudocode definitions of the floating-point operations used by the architecture.

Generation of specific floating-point values

The following pseudocode functions generate specific floating-point values. The `sign` argument of `FPInfinity()`, `FPMaxNormal()`, and `FPZero()` is `'0'` for the positive version and `'1'` for the negative version.

```
// FPZero()
// ========

bits(N) FPZero(bit sign, integer N)
    assert N == 16 || N == 32 || N == 64;
    if N == 16 then
        return sign : '00000 0000000000';
    elsif N == 32 then
        return sign : '00000000 00000000000000000000000';
    else
        return sign : '00000000000 0000000000000000000000000000000000000000000000000';

// FPTwo()
// =======

bits(N) FPTwo(integer N)
    assert N == 32 || N == 64;
    if N == 32 then
        return '0 10000000 00000000000000000000000';
    else
        return '0 10000000000 0000000000000000000000000000000000000000000000000';

// FPThree()
// =========

bits(N) FPThree(integer N)
    assert N == 32 || N == 64;
    if N == 32 then
        return '0 10000000 10000000000000000000000';
    else
        return '0 10000000000 1000000000000000000000000000000000000000000000000';

// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign, integer N)
    assert N == 16 || N == 32 || N == 64;
    if N == 16 then
        return sign : '11110 1111111111';
    elsif N == 32 then
        return sign : '11111110 11111111111111111111111';
    else
        return sign : '11111111110 1111111111111111111111111111111111111111111111111111';
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A2.7.8 浮動小数点演算の擬似コードの詳細

このセクションでは、アーキテクチャによって使用される浮動小数点演算の擬似コードの定義について説明します。

特定の浮動小数点値の生成

以下の擬似コード関数は、特定の浮動小数点値を生成します。`FPInfinity()`、`FPMaxNormal()`、`FPZero()` の符号引数は、正のバージョンの場合は `'0'`、負のバージョンの場合は `'1'` です。

```
// FPZero()
// ========

bits(N) FPZero(bit sign, integer N)
    assert N == 16 || N == 32 || N == 64;
    if N == 16 then
        return sign : '00000 0000000000';
    elsif N == 32 then
        return sign : '00000000 00000000000000000000000';
    else
        return sign : '00000000000 0000000000000000000000000000000000000000000000000';

// FPTwo()
// =======

bits(N) FPTwo(integer N)
    assert N == 32 || N == 64;
    if N == 32 then
        return '0 10000000 00000000000000000000000';
    else
        return '0 10000000000 0000000000000000000000000000000000000000000000000';

// FPThree()
// =========

bits(N) FPThree(integer N)
    assert N == 32 || N == 64;
    if N == 32 then
        return '0 10000000 10000000000000000000000';
    else
        return '0 10000000000 1000000000000000000000000000000000000000000000000';

// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign, integer N)
    assert N == 16 || N == 32 || N == 64;
    if N == 16 then
        return sign : '11110 1111111111';
    elsif N == 32 then
        return sign : '11111110 11111111111111111111111';
    else
        return sign : '11111111110 1111111111111111111111111111111111111111111111111111';
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign, integer N)
    assert N == 16 || N == 32 || N == 64;
    if N == 16 then
        return sign : '11111 0000000000';
    elsif N == 32 then
        return sign : '11111111 00000000000000000000000';
    else
        return sign : '11111111111 0000000000000000000000000000000000000000000000000';

// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN(integer N)
    assert N == 16 || N == 32 || N == 64;
    if N == 16 then
        return '0 11111 1000000000';
    elsif N == 32 then
        return '0 11111111 10000000000000000000000';
    else
        return '0 11111111111 1000000000000000000000000000000000000000000000000';
```

Note

This definition of `FPDefaultNaN()` applies to VFPv3 and VFPv3U. For VFPv2, the sign bit of the result is a single-bit UNKNOWN value, instead of 0.

Negation and absolute value

The floating-point negation and absolute value operations only affect the sign bit. They do not treat NaN operands specially, nor denormalized number operands when flush-to-zero is selected.

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) operand)
    assert N == 32 || N == 64;
    return NOT(operand<N-1>) : operand<N-2:0>;

// FPAbs()
// =======

bits(N) FPAbs(bits(N) operand)
    assert N == 32 || N == 64;
    return '0' : operand<N-2:0>;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign, integer N)
    assert N == 16 || N == 32 || N == 64;
    if N == 16 then
        return sign : '11111 0000000000';
    elsif N == 32 then
        return sign : '11111111 00000000000000000000000';
    else
        return sign : '11111111111 0000000000000000000000000000000000000000000000000';

// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN(integer N)
    assert N == 16 || N == 32 || N == 64;
    if N == 16 then
        return '0 11111 1000000000';
    elsif N == 32 then
        return '0 11111111 10000000000000000000000';
    else
        return '0 11111111111 1000000000000000000000000000000000000000000000000';
```

注

この `FPDefaultNaN()` の定義は、VFPv3 と VFPv3U に適用されます。VFPv2 では、結果の符合ビットは 0 ではなく、シングルビットの未知の値です。

否定と絶対値

浮動小数点の否定と絶対値の演算は、符号ビットだけに影響を与えます。Flush-to-Zero が選択されている場合も、NaN オペランドや非正規化数のオペランドに対して特別な扱いは行われません。

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) operand)
    assert N == 32 || N == 64;
    return NOT(operand<N-1>) : operand<N-2:0>;

// FPAbs()
// =======

bits(N) FPAbs(bits(N) operand)
    assert N == 32 || N == 64;
    return '0' : operand<N-2:0>;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Floating-point value unpacking

The FPUnpack() function determines the type and numerical value of a floating-point number. It also does flush-to-zero processing on input operands.

```
enumeration FPType {FPType_Nonzero, FPType_Zero, FPType_Infinity, FPType_QNaN, FPType_SNaN};

// FPUnpack()
// =========
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in the FPSCR where appropriate.

(FPType, bit, real) FPUnpack(bits(N) fpval, bits(32) fpscr_val)
    assert N == 16 || N == 32 || N == 64;

    if N == 16 then
        sign = fpval<15>;
        exp = fpval<14:10>;
        frac = fpval<9:0>;
        if IsZero(exp) then
            // Produce zero if value is zero
            if IsZero(frac) then
                type = FPType_Zero;  value = 0.0;
            else
                type = FPType_Nonzero;  value = 2^-14 * (UInt(frac) * 2^-10);
        elsif IsOnes(exp) && fpscr_val<26> == '0' then  // Infinity or NaN in IEEE format
            if IsZero(frac) then
                type = FPType_Infinity;  value = 2^1000000;
            else
                type = if frac<9> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;  value = 2^(UInt(exp)-15) * (1.0 + UInt(frac) * 2^-10));

    elsif N == 32 then

        sign = fpval<31>;
        exp  = fpval<30:23>;
        frac = fpval<22:0>;
        if IsZero(exp) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac) || fpscr_val<24> == '1' then
                type = FPType_Zero;  value = 0.0;
                if !IsZero(frac) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpscr_val);
            else
                type = FPType_Nonzero;  value = 2^-126 * (UInt(frac) * 2^-23);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

浮動小数点値のアンパッキング

FPUnpack() 関数は、浮動小数点数のタイプと数値を決定します。入力オペランドに対する Flush-to-Zero 処理も行います。


```
enumeration FPType {FPType_Nonzero, FPType_Zero, FPType_Infinity, FPType_QNaN, FPType_SNaN};

// FPUnpack()
// =========
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in the FPSCR where appropriate.

(FPType, bit, real) FPUnpack(bits(N) fpval, bits(32) fpscr_val)
    assert N == 16 || N == 32 || N == 64;

    if N == 16 then
        sign = fpval<15>;
        exp = fpval<14:10>;
        frac = fpval<9:0>;
        if IsZero(exp) then
            // Produce zero if value is zero
            if IsZero(frac) then
                type = FPType_Zero;  value = 0.0;
            else
                type = FPType_Nonzero;  value = 2^-14 * (UInt(frac) * 2^-10);
        elsif IsOnes(exp) && fpscr_val<26> == '0' then  // Infinity or NaN in IEEE format
            if IsZero(frac) then
                type = FPType_Infinity;  value = 2^1000000;
            else
                type = if frac<9> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;  value = 2^(UInt(exp)-15) * (1.0 + UInt(frac) * 2^-10));

    elsif N == 32 then

        sign = fpval<31>;
        exp  = fpval<30:23>;
        frac = fpval<22:0>;
        if IsZero(exp) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac) || fpscr_val<24> == '1' then
                type = FPType_Zero;  value = 0.0;
                if !IsZero(frac) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpscr_val);
            else
                type = FPType_Nonzero;  value = 2^-126 * (UInt(frac) * 2^-23);
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        elsif IsOnes(exp) then
            if IsZero(frac) then
                type = FPType_Infinity;  value = 2^1000000;
            else
                type = if frac<22> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;  value = 2^(UInt(exp)-127) * (1.0 + UInt(frac) * 2^-23));

    else // N == 64

        sign = fpval<63>;
        exp  = fpval<62:52>;
        frac = fpval<51:0>;
        if IsZero(exp) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac) || fpscr_val<24> == '1' then
                type = FPType_Zero;  value = 0.0;
                if !IsZero(frac) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpscr_val);
            else
                type = FPType_Nonzero;  value = 2^-1022 * (UInt(frac) * 2^-52);
        elsif IsOnes(exp) then
            if IsZero(frac) then
                type = FPType_Infinity;  value = 2^1000000;
            else
                type = if frac<51> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;  value = 2^(UInt(exp)-1023) * (1.0 + UInt(frac) * 2^-52));

    if sign == '1' then value = -value;
    return (type, sign, value);
```

Floating-point exception and NaN handling

The `FPProcessException()` procedure checks whether a floating-point exception is trapped, and handles it accordingly:

```
enumeration FPExc (FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                   FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};

// FPProcessException()
// ====================
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in the FPSCR where appropriate.

FPProcessException(FPExc exception, bits(32) fpscr_val)
    // Get appropriate FPSCR bit numbers
    case exception of
        when FPExc_InvalidOp     enable = 8;   cumul = 0;
        when FPExc_DivideByZero  enable = 9;   cumul = 1;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
        elsif IsOnes(exp) then
            if IsZero(frac) then
                type = FPType_Infinity;  value = 2^1000000;
            else
                type = if frac<22> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;  value = 2^(UInt(exp)-127) * (1.0 + UInt(frac) * 2^-23));

    else // N == 64

        sign = fpval<63>;
        exp  = fpval<62:52>;
        frac = fpval<51:0>;
        if IsZero(exp) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac) || fpscr_val<24> == '1' then
                type = FPType_Zero;  value = 0.0;
                if !IsZero(frac) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpscr_val);
            else
                type = FPType_Nonzero;  value = 2^-1022 * (UInt(frac) * 2^-52);
        elsif IsOnes(exp) then
            if IsZero(frac) then
                type = FPType_Infinity;  value = 2^1000000;
            else
                type = if frac<51> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            type = FPType_Nonzero;  value = 2^(UInt(exp)-1023) * (1.0 + UInt(frac) * 2^-52));

    if sign == '1' then value = -value;
    return (type, sign, value);
```


浮動小数点例外と NaN の処理

`FPProcessException()` プロシージャは、浮動小数点例外がトラップされているかどうかをチェックし、それに応じて処理を行います。


```
enumeration FPExc (FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                   FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};

// FPProcessException()
// ====================
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in the FPSCR where appropriate.

FPProcessException(FPExc exception, bits(32) fpscr_val)
    // Get appropriate FPSCR bit numbers
    case exception of
        when FPExc_InvalidOp     enable = 8;   cumul = 0;
        when FPExc_DivideByZero  enable = 9;   cumul = 1;
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        when FPExc_Overflow      enable = 10;  cumul = 2;
        when FPExc_Underflow     enable = 11;  cumul = 3;
        when FPExc_Inexact       enable = 12;  cumul = 4;
        when FPExc_InputDenorm   enable = 15;  cumul = 7;
    if fpscr_val<enable> then
        IMPLEMENTATION_DEFINED floating-point trap handling;
    else
        FPSCR<cumul> = '1';
    return;
```

The FPProcessNaN() function processes a NaN operand, producing the correct result value and generating an Invalid Operation exception if necessary:

```
// FPProcessNaN()
// ==============
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in the FPSCR where appropriate.

bits(N) FPProcessNaN(FPType type, bits(N) operand, bits(32) fpscr_val)
    assert N == 32 || N == 64;
    topfrac = if N == 32 then 22 else 51;
    result = operand;
    if type = FPType_SNaN then
        result<topfrac> = '1';
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    if fpscr_val<25> == '1' then  // DefaultNaN requested
        result = FPDefaultNaN(N);
    return result;
```

The FPProcessNaNs() function performs the standard NaN processing for a two-operand operation:

```
// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in the FPSCR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
                                 bits(N) op1, bits(N) op2,
                                 bits(32) fpscr_val)
    assert N == 32 || N == 64;
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpscr_val);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpscr_val);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpscr_val);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpscr_val);
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        when FPExc_Overflow      enable = 10;  cumul = 2;
        when FPExc_Underflow     enable = 11;  cumul = 3;
        when FPExc_Inexact       enable = 12;  cumul = 4;
        when FPExc_InputDenorm   enable = 15;  cumul = 7;
    if fpscr_val<enable> then
        IMPLEMENTATION_DEFINED floating-point trap handling;
    else
        FPSCR<cumul> = '1';
    return;
```


FPProcessNaN() 関数は、NaN オペランドを処理し、必要に応じて正しい結果の値と無効演算例外を生成します。


```
// FPProcessNaN()
// ==============
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in the FPSCR where appropriate.

bits(N) FPProcessNaN(FPType type, bits(N) operand, bits(32) fpscr_val)
    assert N == 32 || N == 64;
    topfrac = if N == 32 then 22 else 51;
    result = operand;
    if type = FPType_SNaN then
        result<topfrac> = '1';
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    if fpscr_val<25> == '1' then  // DefaultNaN requested
        result = FPDefaultNaN(N);
    return result;
```


FPProcessNaNs() 関数は、2 オペランドの動作では標準の NaN 処理を実行します。


```
// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in the FPSCR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
                                 bits(N) op1, bits(N) op2,
                                 bits(32) fpscr_val)
    assert N == 32 || N == 64;
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpscr_val);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpscr_val);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpscr_val);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpscr_val);
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result
    return (done, result);
```

Floating-point rounding

The FPRound() function rounds and encodes a floating-point result value to a specified destination format. This includes processing Overflow, Underflow and Inexact floating-point exceptions and performing flush-to-zero processing on result values.

```
// FPRound()
// =========
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in the FPSCR where appropriate.

bits(N) FPRound(real result, integer N, bits(32) fpscr_val)
    assert N == 16 || N == 32 || N == 64;
    assert result != 0.0;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else  // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    if result < 0.0 then
        sign = '1';  mantissa = -result;
    else
        sign = '0';  mantissa = result;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Deal with flush-to-zero.
    if fpscr_val<24> == '1' && N != 16 && exponent < minimum_exp then
        result = FPZero(sign, N);
        FPSCR.UFC = '1';  // Flush-to-zero never generates a trapped exception
    else

        // Start creating the exponent value for the result. Start by biasing the actual exponent
        // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
        biased_exp = Max(exponent - minimum_exp + 1, 0);
        if biased_exp == 0 then mantissa = mantissa / 2^(minimum_exp - exponent);

        // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
        int_mant = RoundDown(mantissa * 2^F);  // < 2^F if biased_exp == 0, >= 2^F if not
        error = mantissa * 2^F - int_mant;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result
    return (done, result);
```

浮動小数点数の丸め

FPRound() 関数は、浮動小数点の結果の値を指定されたデスティネーションフォーマットに丸め、エンコードします。この関数には、オーバフロー、アンダーフロー、不正確の各浮動小数点例外の処理、および結果の値に対する Flush-to-Zero 処理の実行が含まれます。

```
// FPRound()
// =========
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in the FPSCR where appropriate.

bits(N) FPRound(real result, integer N, bits(32) fpscr_val)
    assert N == 16 || N == 32 || N == 64;
    assert result != 0.0;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else  // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    if result < 0.0 then
        sign = '1';  mantissa = -result;
    else
        sign = '0';  mantissa = result;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Deal with flush-to-zero.
    if fpscr_val<24> == '1' && N != 16 && exponent < minimum_exp then
        result = FPZero(sign, N);
        FPSCR.UFC = '1';  // Flush-to-zero never generates a trapped exception
    else

        // Start creating the exponent value for the result. Start by biasing the actual exponent
        // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
        biased_exp = Max(exponent - minimum_exp + 1, 0);
        if biased_exp == 0 then mantissa = mantissa / 2^(minimum_exp - exponent);

        // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
        int_mant = RoundDown(mantissa * 2^F);  // < 2^F if biased_exp == 0, >= 2^F if not
        error = mantissa * 2^F - int_mant;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        // Underflow occurs if exponent is too small before rounding, and result is inexact or
        // the Underflow exception is trapped.
        if biased_exp == 0 && (error != 0.0 || fpscr_val<11> == '1') then
            FPProcessException(FPExc_Underflow, fpscr_val);

        // Round result according to rounding mode.
        case fpscr_val<23:22> of
            when '00'  // Round to Nearest (rounding to even if exactly halfway)
                round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
                overflow_to_inf = TRUE;
            when '01'  // Round towards Plus Infinity
                round_up = (error != 0.0 && sign == '0');
                overflow_to_inf = (sign == '0');
            when '10'  // Round towards Minus Infinity
                round_up = (error != 0.0 && sign == '1');
                overflow_to_inf = (sign == '1');
            when '11'  // Round towards Zero
                round_up = FALSE;
                overflow_to_inf = FALSE;
        if round_up then
            int_mant = int_mant + 1;
            if int_mant == 2^F then      // Rounded up from denormalized to normalized
                biased_exp = 1;
            if int_mant == 2^(F+1) then  // Rounded up to next exponent
                biased_exp = biased_exp + 1;  int_mant = int_mant DIV 2;

        // Deal with overflow and generate result.
        if N != 16 || fpscr_val<26> == '0' then  // Single, double or IEEE half precision
            if biased_exp >= 2^E - 1 then
                result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
                FPProcessException(FPExc_Overflow, fpscr_val);
            else
                result = sign : biased_exp<E-1:0> : int_mant<F-1:0>;
        else                                     // Alternative half precision
            if biased_exp >= 2^E then
                result = sign : Ones(15);
                FPProcessException(FPExc_InvalidOp, fpscr_val);
                error = 0.0; // avoid an Inexact exception
            else
                result = sign : biased_exp<E-1:0> : int_mant<F-1:0>;

        // Deal with Inexact exception.
        if error != 0 then
            FPProcessException(FPExc_Inexact, fpscr_val);

    return result;
```

Selection of ARM standard floating-point arithmetic

`StandardFPSCRValue` is an FPSCR value that selects ARM standard floating-point arithmetic. Most of the arithmetic functions have a boolean `fpscr_controlled` argument that is `TRUE` for VFP operations and `FALSE` for Advanced SIMD operations, and that selects between using the real FPSCR value and this value.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
        // Underflow occurs if exponent is too small before rounding, and result is inexact or
        // the Underflow exception is trapped.
        if biased_exp == 0 && (error != 0.0 || fpscr_val<11> == '1') then
            FPProcessException(FPExc_Underflow, fpscr_val);

        // Round result according to rounding mode.
        case fpscr_val<23:22> of
            when '00'  // Round to Nearest (rounding to even if exactly halfway)
                round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
                overflow_to_inf = TRUE;
            when '01'  // Round towards Plus Infinity
                round_up = (error != 0.0 && sign == '0');
                overflow_to_inf = (sign == '0');
            when '10'  // Round towards Minus Infinity
                round_up = (error != 0.0 && sign == '1');
                overflow_to_inf = (sign == '1');
            when '11'  // Round towards Zero
                round_up = FALSE;
                overflow_to_inf = FALSE;
        if round_up then
            int_mant = int_mant + 1;
            if int_mant == 2^F then      // Rounded up from denormalized to normalized
                biased_exp = 1;
            if int_mant == 2^(F+1) then  // Rounded up to next exponent
                biased_exp = biased_exp + 1;  int_mant = int_mant DIV 2;

        // Deal with overflow and generate result.
        if N != 16 || fpscr_val<26> == '0' then  // Single, double or IEEE half precision
            if biased_exp >= 2^E - 1 then
                result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
                FPProcessException(FPExc_Overflow, fpscr_val);
            else
                result = sign : biased_exp<E-1:0> : int_mant<F-1:0>;
        else                                     // Alternative half precision
            if biased_exp >= 2^E then
                result = sign : Ones(15);
                FPProcessException(FPExc_InvalidOp, fpscr_val);
                error = 0.0; // avoid an Inexact exception
            else
                result = sign : biased_exp<E-1:0> : int_mant<F-1:0>;

        // Deal with Inexact exception.
        if error != 0 then
            FPProcessException(FPExc_Inexact, fpscr_val);

    return result;
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
// StandardFPSCRValue()
// ====================

bits(32) StandardFPSCRValue()
    return '00000' : FPSCR<26> : '11000000000000000000000000';
```

Comparisons

The FPCompare() function compares two floating-point numbers, producing an (N,Z,C,V) flags result as shown in Table A2-8:

Table A2-8 VFP comparison flag values

Comparison result	N	Z	C	V
Equal	0	1	1	0
Less than	1	0	0	0
Greater than	0	0	1	0
Unordered	0	0	1	1

This result is used to define the VCMP instruction in the VFP extension. The VCMP instruction writes these flag values in the FPSCR. After using a VMRS instruction to transfer them to the APSR, they can be used to control conditional execution as shown in Table A8-1 on page A8-8.

```
// FPCompare()
// ===========

(bit, bit, bit, bit) FPCompare(bits(N) op1, bits(N) op2, boolean quiet_nan_exc,
                               boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = ('0','0','1','1');
        if type1==FPType_SNaN || type2==FPType_SNaN || quiet_nan_exc then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = ('0','1','1','0');
        elsif value1 < value2 then
            result = ('1','0','0','0');
        else  // value1 > value2
            result = ('0','0','1','0');
    return result;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARM 標準の浮動小数点算術演算の選択

StandardFPSCRValue は、ARM 標準の浮動小数点算術演算を選択する FPSCR 値です。ほとんどの算術演算関数には、ブールの fpscr_controlled 引数があります。これは、VFP 演算の場合は TRUE、アドバンスト SIMD 演算の場合は FALSE で、実際の FPSCR 値とこの値のどちらを使用するかを選択します。

```
// StandardFPSCRValue()
// ====================

bits(32) StandardFPSCRValue()
    return '00000' : FPSCR<26> : '11000000000000000000000000';
```

比較

FPCompare() 関数は、2 つの浮動小数点数を比較し、表 A2-8 に示すような (N、Z、C、V) フラグの結果を生成します。

表 A2-8 VFP 比較フラグの値

比較結果	N	Z	C	V
等しい	0	1	1	0
<	1	0	0	0
>	0	0	1	0
規則なし	0	0	1	1

この結果は、VFP 拡張機能の VCMP 命令を定義するために使用されます。VCMP 命令は、これらのフラグの値を FPSCR に書き込みます。VMRS 命令を使用して APSR にフラグの値を転送した後は、P. A8-8 の表 A8-1 に示すように条件実行の制御に使用できます。

```
// FPCompare()
// ===========

(bit, bit, bit, bit) FPCompare(bits(N) op1, bits(N) op2, boolean quiet_nan_exc,
                               boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = ('0','0','1','1');
        if type1==FPType_SNaN || type2==FPType_SNaN || quiet_nan_exc then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = ('0','1','1','0');
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The FPCompareEQ(), FPCompareGE() and FPCompareGT() functions are used to describe Advanced SIMD instructions that perform floating-point comparisons.

```
// FPCompareEQ()
// ============

boolean FPCompareEQ(bits(32) op1, bits(32) op2, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        if type1==FPType_SNaN || type2==FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
    return result;

// FPCompareGE()
// ============

boolean FPCompareGE(bits(32) op1, bits(32) op2, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
    return result;

// FPCompareGT()
// ============

boolean FPCompareGT(bits(32) op1, bits(32) op2, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
    return result;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        elsif value1 < value2 then
            result = ('1','0','0','0');
        else  // value1 > value2
            result = ('0','0','1','0');
    return result;
```

浮動小数点比較を実行するアドバンスト SIMD 命令を記述するには、FPCompareEQ()、FPCompareGE()、FPCompareGT() 関数が使用されます。

```
// FPCompareEQ()
// ============

boolean FPCompareEQ(bits(32) op1, bits(32) op2, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        if type1==FPType_SNaN || type2==FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
    return result;

// FPCompareGE()
// ============

boolean FPCompareGE(bits(32) op1, bits(32) op2, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
    return result;

// FPCompareGT()
// ============

boolean FPCompareGT(bits(32) op1, bits(32) op2, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
    return result;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Maximum and minimum

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
    if !done then
        if type1 == FPType_Zero && type2 == FPType_Zero && sign1 == NOT(sign2) then
            // Opposite-signed zeros produce +0.0
            result = FPZero('0', N);
        else
            // All other cases can be evaluated on the values produced by FPUnpack()
            result = if value1 > value2 then op1 else op2;
    return result;

// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
    if !done then
        if type1 == FPType_Zero && type2 == FPType_Zero && sign1 == NOT(sign2) then
            // Opposite-signed zeros produce -0.0
            result = FPZero('1', N);
        else
            // All other cases can be evaluated on the values produced by FPUnpack()
            result = if value1 < value2 then op1 else op2;
    return result;
```

Addition and subtraction

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

最大値と最小値


```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
    if !done then
        if type1 == FPType_Zero && type2 == FPType_Zero && sign1 == NOT(sign2) then
            // Opposite-signed zeros produce +0.0
            result = FPZero('0', N);
        else
            // All other cases can be evaluated on the values produced by FPUnpack()
            result = if value1 > value2 then op1 else op2;
    return result;

// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
    if !done then
        if type1 == FPType_Zero && type2 == FPType_Zero && sign1 == NOT(sign2) then
            // Opposite-signed zeros produce -0.0
            result = FPZero('1', N);
        else
            // All other cases can be evaluated on the values produced by FPUnpack()
            result = if value1 < value2 then op1 else op2;
    return result;
```


加算と減算


```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
                result = FPDefaultNaN(N);
                FPProcessException(FPExc_InvalidOp, fpscr_val);
            elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
                result = FPInfinity('0', N);
            elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
                result = FPInfinity('1', N);
            elsif zero1 && zero2 && sign1 == sign2 then
                result = FPZero(sign1, N);
            else
                result_value = value1 + value2;
                if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                    result_sign = if fpscr_val<23:22> == '10' then '1' else '0';
                    result = FPZero(result_sign, N);
                else
                    result = FPRound(result_value, N, fpscr_val);
        return result;

    // FPSub()
    // =======

    bits(N) FPSub(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
        assert N == 32 || N == 64;
        fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
        (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
        (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
        (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
        if !done then
            inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
            zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
            if inf1 && inf2 && sign1 == sign2 then
                result = FPDefaultNaN(N);
                FPProcessException(FPExc_InvalidOp, fpscr_val);
            elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
                result = FPInfinity('0', N);
            elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
                result = FPInfinity('1', N);
            elsif zero1 && zero2 && sign1 == NOT(sign2) then
                result = FPZero(sign1, N);
            else
                result_value = value1 - value2;
                if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                    result_sign = if fpscr_val<23:22> == '10' then '1' else '0';
                    result = FPZero(result_sign, N);
                else
                    result = FPRound(result_value, N, fpscr_val);
        return result;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
                result = FPDefaultNaN(N);
                FPProcessException(FPExc_InvalidOp, fpscr_val);
            elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
                result = FPInfinity('0', N);
            elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
                result = FPInfinity('1', N);
            elsif zero1 && zero2 && sign1 == sign2 then
                result = FPZero(sign1, N);
            else
                result_value = value1 + value2;
                if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                    result_sign = if fpscr_val<23:22> == '10' then '1' else '0';
                    result = FPZero(result_sign, N);
                else
                    result = FPRound(result_value, N, fpscr_val);
        return result;

    // FPSub()
    // =======

    bits(N) FPSub(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
        assert N == 32 || N == 64;
        fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
        (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
        (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
        (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
        if !done then
            inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
            zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
            if inf1 && inf2 && sign1 == sign2 then
                result = FPDefaultNaN(N);
                FPProcessException(FPExc_InvalidOp, fpscr_val);
            elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
                result = FPInfinity('0', N);
            elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
                result = FPInfinity('1', N);
            elsif zero1 && zero2 && sign1 == NOT(sign2) then
                result = FPZero(sign1, N);
            else
                result_value = value1 - value2;
                if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                    result_sign = if fpscr_val<23:22> == '10' then '1' else '0';
                    result = FPZero(result_sign, N);
                else
                    result = FPRound(result_value, N, fpscr_val);
        return result;
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Multiplication and division

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(N);
            FPProcessException(FPExc_InvalidOp, fpscr_val);
        elsif inf1 || inf2 then
            result_sign = if sign1 == sign2 then '0' else '1';
            result = FPInfinity(result_sign, N);
        elsif zero1 || zero2 then
            result_sign = if sign1 == sign2 then '0' else '1';
            result = FPZero(result_sign, N);
        else
            result = FPRound(value1*value2, N, fpscr_val);
    return result;

// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN(N);
            FPProcessException(FPExc_InvalidOp, fpscr_val);
        elsif inf1 || zero2 then
            result_sign = if sign1 == sign2 then '0' else '1';
            result = FPInfinity(result_sign, N);
            if !inf1 then FPProcessException(FPExc_DivideByZero);
        elsif zero1 || inf2 then
            result_sign = if sign1 == sign2 then '0' else '1';
            result = FPZero(result_sign, N);
        else
            result = FPRound(value1/value2, N, fpscr_val);
    return result;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

乗算と除算

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(N);
            FPProcessException(FPExc_InvalidOp, fpscr_val);
        elsif inf1 || inf2 then
            result_sign = if sign1 == sign2 then '0' else '1';
            result = FPInfinity(result_sign, N);
        elsif zero1 || zero2 then
            result_sign = if sign1 == sign2 then '0' else '1';
            result = FPZero(result_sign, N);
        else
            result = FPRound(value1*value2, N, fpscr_val);
    return result;

// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN(N);
            FPProcessException(FPExc_InvalidOp, fpscr_val);
        elsif inf1 || zero2 then
            result_sign = if sign1 == sign2 then '0' else '1';
            result = FPInfinity(result_sign, N);
            if !inf1 then FPProcessException(FPExc_DivideByZero);
        elsif zero1 || inf2 then
            result_sign = if sign1 == sign2 then '0' else '1';
            result = FPZero(result_sign, N);
        else
            result = FPRound(value1/value2, N, fpscr_val);
    return result;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Reciprocal estimate and step

The Advanced SIMD extension includes instructions that support Newton-Raphson calculation of the reciprocal of a number.

The VRECPE instruction produces the initial estimate of the reciprocal. It uses the following pseudocode functions:

```
// FPRecipEstimate()
// =================

bits(32) FPRecipEstimate(bits(32) operand)

    (type,sign,value) = FPUnpack(operand, StandardFPSCRValue());
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, StandardFPSCRValue());
    elsif type = FPType_Infinity then
        result = FPZero(sign, 32);
    elsif type = FPType_Zero then
        result = FPInfinity(sign, 32);
        FPProcessException(FPExc_DivideByZero, StandardFPSCRValue());
    elsif Abs(value) >= 2^126 then  // Result underflows to zero of correct sign
        result = FPZero(sign, 32);
        FPProcessException(FPExc_Underflow, StandardFPSCRValue());;
    else
        // Operand must be normalized, since denormalized numbers are flushed to zero. Scale to a
        // double-precision value in the range 0.5 <= x < 1.0, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 = double-precision biased version of
        // -1, fraction = original fraction extended with zeros.
        scaled = operand<31> : '01111111110' : operand<22:0> : Zeros(29);
        result_exp = 253 - UInt(operand<30:23>);   // In range 253-252 = 1 to 253-1 = 252

        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_estimate(scaled);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256. Convert
        // to scaled single-precision result with copied sign bit and high-order fraction bits,
        // and exponent calculated above.
        result = estimate<63> : result_exp<7:0> : estimate<51:29>;

    return result;

// UnsignedRecipEstimate()
// =======================

bits(32) UnsignedRecipEstimate(bits(32) operand)

    if operand<31> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(32);
    else
        // Generate double-precision value = operand * 2^-32. This has zero sign bit,
        // exponent = 1022 = double-precision biased version of -1, fraction taken from
        // operand, excluding its most significant bit.
        dp_operand = '0 01111111110' : operand<30:0> : Zeros(21);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

逆数概算とステップ

アドバンスト SIMD 拡張機能には、ニュートン・ラプソン反復法によって逆数を計算する命令が含まれています。

VRECPE 命令は、逆数の初期概算値を生成します。この命令は、次の擬似コード関数を使用します。

```
// FPRecipEstimate()
// =================

bits(32) FPRecipEstimate(bits(32) operand)

    (type,sign,value) = FPUnpack(operand, StandardFPSCRValue());
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, StandardFPSCRValue());
    elsif type = FPType_Infinity then
        result = FPZero(sign, 32);
    elsif type = FPType_Zero then
        result = FPInfinity(sign, 32);
        FPProcessException(FPExc_DivideByZero, StandardFPSCRValue());
    elsif Abs(value) >= 2^126 then  // Result underflows to zero of correct sign
        result = FPZero(sign, 32);
        FPProcessException(FPExc_Underflow, StandardFPSCRValue());;
    else
        // Operand must be normalized, since denormalized numbers are flushed to zero. Scale to a
        // double-precision value in the range 0.5 <= x < 1.0, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 = double-precision biased version of
        // -1, fraction = original fraction extended with zeros.
        scaled = operand<31> : '01111111110' : operand<22:0> : Zeros(29);
        result_exp = 253 - UInt(operand<30:23>);   // In range 253-252 = 1 to 253-1 = 252

        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_estimate(scaled);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256. Convert
        // to scaled single-precision result with copied sign bit and high-order fraction bits,
        // and exponent calculated above.
        result = estimate<63> : result_exp<7:0> : estimate<51:29>;

    return result;

// UnsignedRecipEstimate()
// =======================

bits(32) UnsignedRecipEstimate(bits(32) operand)

    if operand<31> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(32);
    else
        // Generate double-precision value = operand * 2^-32. This has zero sign bit,
        // exponent = 1022 = double-precision biased version of -1, fraction taken from
        // operand, excluding its most significant bit.
        dp_operand = '0 01111111110' : operand<30:0> : Zeros(21);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_estimate(dp_operand);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Multiply by 2^31 and convert to an unsigned integer - this just involves
        // concatenating the implicit units bit with the top 31 fraction bits.
        result = '1' : estimate<51:21>;

    return result;
```

where `recip_estimate()` is defined by the following C function:

```
double recip_estimate(double a)
{
  int q, s;
  double r;
  q = (int)(a * 512.0);         /* a in units of 1/512 rounded down */
  r = 1.0 / (((double)q + 0.5) / 512.0);  /* reciprocal r */
  s = (int)(256.0 * r + 0.5);   /* r in units of 1/256 rounded to nearest */
  return (double)s / 256.0;
}
```

Table A2-9 shows the results where input values are out of range.

Table A2-9 VRECPE results for out-of-range inputs

Number type	Input Vm[i]	Result Vd[i]
Integer	<= 0x7FFFFFFF	0xFFFFFFFF
Floating-point	NaN	Default NaN
Floating-point	+/– 0 or denormalized number	+/– Infinity [a]
Floating-point	+/– infinity	+/– 0
Floating-point	Absolute value >= 2^{126}	+/– 0

a. The Division by Zero exception bit in the FPSCR (FPSCR[1]) is set

The Newton-Raphson iteration:

$x_{n+1} = x_n(2-dx_n)$

converges to $(1/d)$ if x_0 is the result of `VRECPE` applied to d.

The VRECPS instruction performs a 2 - op1*op2 calculation and can be used with a multiplication to perform a step of this iteration. The functionality of this instruction is defined by the following pseudocode function:

```
// FPRecipStep()
// =============
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_estimate(dp_operand);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Multiply by 2^31 and convert to an unsigned integer - this just involves
        // concatenating the implicit units bit with the top 31 fraction bits.
        result = '1' : estimate<51:21>;

    return result;
```

ここで、`recip_estimate()` は次の C 関数によって定義されます。

```
double recip_estimate(double a)
{
  int q, s;
  double r;
  q = (int)(a * 512.0);         /* a in units of 1/512 rounded down */
  r = 1.0 / (((double)q + 0.5) / 512.0);  /* reciprocal r */
  s = (int)(256.0 * r + 0.5);   /* r in units of 1/256 rounded to nearest */
  return (double)s / 256.0;
}
```

入力値が範囲外の場合の結果を、表 A2-9 に示します。

表 A2-9 範囲外の入力に対する VRECPE の結果

数値の型	入力、Vm[i]	結果、Vd[i]
整数	<= 0x7FFFFFFF	0xFFFFFFFF
浮動小数点数	NaN	デフォルト NaN
浮動小数点数	+/- 0 または非正規化数	+/- 無限大 [a]
浮動小数点数	+/- 無限大	+/- 0
浮動小数点数	絶対値 >= 2^{126}	+/- 0

a. FPSCR (FPSCR[1]) にある、0による除算例外ビットがセットされます。

ニュートン・ラプソン反復法

$x_{n+1} = x_n(2 - dxn)$

は、d に `VRECPE` を適用した結果が x_0 のとき、$(1 / d)$ に収束します。

VRECPS 命令は、2 − op1 * op2 の計算を実行し、乗算と組み合わせることで、反復の 1 ステップを実行できます。この命令の機能は、次の擬似コード関数で定義されます。

```
// FPRecipStep()
// =============
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
bits(32) FPRecipStep(bits(32) op1, bits(32) op2)
    (type1,sign1,value1) = FPUnpack(op1, StandardFPSCRValue());
    (type2,sign2,value2) = FPUnpack(op2, StandardFPSCRValue());
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, StandardFPSCRValue());
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', 32);
        else
            product = FPMul(op1, op2, FALSE);
        result = FPSub(FPTwo(32), product, FALSE);
    return result;
```

Table A2-10 shows the results where input values are out of range.

Table A2-10 VRECPS results for out-of-range inputs

Input Vn[i]	Input Vm[i]	Result Vd[i]
Any NaN	-	Default NaN
-	Any NaN	Default NaN
+/– 0.0 or denormalized number	+/– infinity	2.0
+/– infinity	+/– 0.0 or denormalized number	2.0

Square root

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) operand, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type,sign,value) = FPUnpack(operand, fpscr_val);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, fpscr_val);
    elsif type == FPType_Zero || (type = FPType_Infinity && sign == '0') then
        result = operand;
    elsif sign == '1' then
        result = FPDefaultNaN(N);
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    else
        result = FPRound(Sqrt(value), N, fpscr_val);
    return result;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

```
bits(32) FPRecipStep(bits(32) op1, bits(32) op2)
    (type1,sign1,value1) = FPUnpack(op1, StandardFPSCRValue());
    (type2,sign2,value2) = FPUnpack(op2, StandardFPSCRValue());
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, StandardFPSCRValue());
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', 32);
        else
            product = FPMul(op1, op2, FALSE);
        result = FPSub(FPTwo(32), product, FALSE);
    return result;
```

入力値が範囲外の場合の結果を、表 A2- 10 に示します。

表 A2- 10 範囲外の入力に対する VRECPS の結果

入力、Vn[i]	入力、Vm[i]	結果、Vd[i]
すべての NaN	–	デフォルト NaN
–	すべての NaN	デフォルト NaN
+/- 0.0 または非正規化数	+/- 無限大	2.0
+/- 無限大	+/- 0.0 または非正規化数	2.0

平方根

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) operand, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type,sign,value) = FPUnpack(operand, fpscr_val);
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, fpscr_val);
    elsif type == FPType_Zero || (type = FPType_Infinity && sign == '0') then
        result = operand;
    elsif sign == '1' then
        result = FPDefaultNaN(N);
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    else
        result = FPRound(Sqrt(value), N, fpscr_val);
    return result;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Reciprocal square root

The Advanced SIMD extension includes instructions that support Newton-Raphson calculation of the reciprocal of the square root of a number.

The VRSQRTE instruction produces the initial estimate of the reciprocal of the square root. It uses the following pseudocode functions:

```
// FPRSqrtEstimate()
// ================

bits(32) FPRSqrtEstimate(bits(32) operand)

    (type,sign,value) = FPUnpack(operand, StandardFPSCRValue());
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, StandardFPSCRValue());
    elsif type = FPType_Zero then
        result = FPInfinity(sign, 32);
        FPProcessException(FPExc_DivideByZero, StandardFPSCRValue());
    elsif sign == '1' then
        result = FPDefaultNaN(32);
        FPProcessException(FPExc_InvalidOp, StandardFPSCRValue());
    elsif type = FPType_Infinity then
        result = FPZero('0', 32);
    else
        // Operand must be normalized, since denormalized numbers are flushed to zero. Scale to a
        // double-precision value in the range 0.25 <= x < 1.0, with the evenness or oddness of
        // the exponent unchanged, and calculate result exponent. Scaled value has copied sign
        // bit, exponent = 1022 or 1021 = double-precision biased version of -1 or -2, fraction
        // = original fraction extended with zeros.
        if operand<23> == '0' then
          scaled = operand<31> : '01111111110' : operand<22:0> : Zeros(29);
        else
          scaled = operand<31> : '01111111101' : operand<22:0> : Zeros(29);
        result_exp = (380 - UInt(operand<30:23>)) DIV 2;

        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_sqrt_estimate(scaled);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256. Convert
        // to scaled single-precision result with copied sign bit and high-order fraction bits,
        // and exponent calculated above.
        result = estimate<63> : result_exp<7:0> : estimate<51:29>;

    return result;

// UnsignedRSqrtEstimate()
// =======================

bits(32) UnsignedRSqrtEstimate(bits(32) operand)

    if operand<31:30> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(32);
    else
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

逆数平方根

アドバンスト SIMD 拡張機能には、ニュートン・ラプソン反復法によって数値の平方根の逆数を計算する命令が含まれています。

VRSQRTE 命令は、逆数平方根の初期概算値を生成します。この命令は、次の擬似コード関数を使用します。

```
// FPRSqrtEstimate()
// ================

bits(32) FPRSqrtEstimate(bits(32) operand)

    (type,sign,value) = FPUnpack(operand, StandardFPSCRValue());
    if type == FPType_SNaN || type == FPType_QNaN then
        result = FPProcessNaN(type, operand, StandardFPSCRValue());
    elsif type = FPType_Zero then
        result = FPInfinity(sign, 32);
        FPProcessException(FPExc_DivideByZero, StandardFPSCRValue());
    elsif sign == '1' then
        result = FPDefaultNaN(32);
        FPProcessException(FPExc_InvalidOp, StandardFPSCRValue());
    elsif type = FPType_Infinity then
        result = FPZero('0', 32);
    else
        // Operand must be normalized, since denormalized numbers are flushed to zero. Scale to a
        // double-precision value in the range 0.25 <= x < 1.0, with the evenness or oddness of
        // the exponent unchanged, and calculate result exponent. Scaled value has copied sign
        // bit, exponent = 1022 or 1021 = double-precision biased version of -1 or -2, fraction
        // = original fraction extended with zeros.
        if operand<23> == '0' then
          scaled = operand<31> : '01111111110' : operand<22:0> : Zeros(29);
        else
          scaled = operand<31> : '01111111101' : operand<22:0> : Zeros(29);
        result_exp = (380 - UInt(operand<30:23>)) DIV 2;

        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_sqrt_estimate(scaled);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256. Convert
        // to scaled single-precision result with copied sign bit and high-order fraction bits,
        // and exponent calculated above.
        result = estimate<63> : result_exp<7:0> : estimate<51:29>;

    return result;

// UnsignedRSqrtEstimate()
// =======================

bits(32) UnsignedRSqrtEstimate(bits(32) operand)

    if operand<31:30> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(32);
    else
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        // Generate double-precision value = operand * 2^-32. This has zero sign bit,
        // exponent = 1022 or 1021 = double-precision biased version of -1 or -2,
        // fraction taken from operand, excluding its most significant one or two bits.
        if operand<31> == '1' then
            dp_operand = '0 01111111110' : operand<30:0> : Zeros(21);
        else  // operand<31:30> == '01'
            dp_operand = '0 01111111101' : operand<29:0> : Zeros(22);

        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_sqrt_estimate(dp_operand);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Multiply by 2^31 and convert to an unsigned integer - this just involves
        // concatenating the implicit units bit with the top 31 fraction bits.
        result = '1' : estimate<51:21>;

    return result;
```

where `recip_sqrt_estimate()` is defined by the following C function:

```
double recip_sqrt_estimate(double a)
{
  int q0, q1, s;
  double r;
  if (a < 0.5)  /* range 0.25 <= a < 0.5 */
  {
    q0 = (int)(a * 512.0);       /* a in units of 1/512 rounded down */
    r = 1.0 / sqrt(((double)q0 + 0.5) / 512.0); /* reciprocal root r */
  }
  else          /* range 0.5  <= a < 1.0 */
  {
    q1 = (int)(a * 256.0);       /* a in units of 1/256 rounded down */
    r = 1.0 / sqrt(((double)q1 + 0.5) / 256.0); /* reciprocal root r */
  }
  s = (int)(256.0 * r + 0.5);  /* r in units of 1/256 rounded to nearest */
  return (double)s / 256.0;
}
```

Table A2-11 shows the results where input values are out of range.

Table A2-11 VRSQRTE results for out-of-range inputs

Number type	Input Vm[i]	Result Vd[i]
Integer	<= 0x3FFFFFFF	0xFFFFFFFF
Floating-point	NaN, – normalized number, – infinity	Default NaN
Floating-point	– 0 or – denormalized number	– infinity [a]
Floating-point	+ 0 or + denormalized number	+ infinity [a]
Floating-point	+ infinity	+ 0

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

```
        // Generate double-precision value = operand * 2^-32. This has zero sign bit,
        // exponent = 1022 or 1021 = double-precision biased version of -1 or -2,
        // fraction taken from operand, excluding its most significant one or two bits.
        if operand<31> == '1' then
            dp_operand = '0 01111111110' : operand<30:0> : Zeros(21);
        else  // operand<31:30> == '01'
            dp_operand = '0 01111111101' : operand<29:0> : Zeros(22);

        // Call C function to get reciprocal estimate of scaled value.
        estimate = recip_sqrt_estimate(dp_operand);

        // Result is double-precision and a multiple of 1/256 in the range 1 to 511/256.
        // Multiply by 2^31 and convert to an unsigned integer - this just involves
        // concatenating the implicit units bit with the top 31 fraction bits.
        result = '1' : estimate<51:21>;

    return result;
```

ここで、`recip_sqrt_estimate()` は次の C 関数によって定義されます。

```
double recip_sqrt_estimate(double a)
{
  int q0, q1, s;
  double r;
  if (a < 0.5)  /* range 0.25 <= a < 0.5 */
  {
    q0 = (int)(a * 512.0);       /* a in units of 1/512 rounded down */
    r = 1.0 / sqrt(((double)q0 + 0.5) / 512.0); /* reciprocal root r */
  }
  else          /* range 0.5  <= a < 1.0 */
  {
    q1 = (int)(a * 256.0);       /* a in units of 1/256 rounded down */
    r = 1.0 / sqrt(((double)q1 + 0.5) / 256.0); /* reciprocal root r */
  }
  s = (int)(256.0 * r + 0.5);  /* r in units of 1/256 rounded to nearest */
  return (double)s / 256.0;
}
```

入力値が範囲外の場合の結果を、表 A2-11 に示します。

表 A2-11 範囲外の入力に対する VRSQRTE の結果

数値の型	入力、Vm[i]	結果、Vd[i]
整数	<= 0x3FFFFFFF	0xFFFFFFFF
浮動小数点数	NaN、- 正規化数、- 無限大	デフォルト NaN
浮動小数点数	- 0 または - 非正規化数	- 無限大 [a]
浮動小数点数	+ 0 または + 非正規化数	+ 無限大 [a]
浮動小数点数	+ 無限大	+ 0

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

a. The Division by Zero exception bit in the FPSCR (FPSCR[1]) is set.

The Newton-Raphson iteration:

$x_{n+1} = x_n(3-dx_n^2)/2$

converges to $(1/\sqrt{d})$ if x_0 is the result of `VRSQRTE` applied to d.

The `VRSQRTS` instruction performs a (3 – op1*op2)/2 calculation and can be used with two multiplications to perform a step of this iteration. The functionality of this instruction is defined by the following pseudocode function:

```
// FPRSqrtStep()
// =============

bits(32) FPRSqrtStep(bits(32) op1, bits(32) op2)
    (type1,sign1,value1) = FPUnpack(op1, StandardFPSCRValue());
    (type2,sign2,value2) = FPUnpack(op2, StandardFPSCRValue());
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, StandardFPSCRValue());
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', 32);
        else
            product = FPMul(op1, op2, FALSE);
        result = FPDiv(FPSub(FPThree(32), product, FALSE), FPTwo(32), FALSE);
    return result;
```

Table A2-12 shows the results where input values are out of range.

Table A2-12 VRSQRTS results for out-of-range inputs

Input Vn[i]	Input Vm[i]	Result Vd[i]
Any NaN	-	Default NaN
-	Any NaN	Default NaN
+/– 0.0 or denormalized number	+/– infinity	1.5
+/– infinity	+/– 0.0 or denormalized number	1.5

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

a. FPSCR (FPSCR[1]) にある、0 による除算例外ビットがセットされます。

ニュートン・ラプソン反復法

$x_{n+1} = xn(3 - dxn^2) / 2$

は、d に `VRSQRTE` を適用した結果が x_0 のとき、$(1/\sqrt{d})$ に収束します。

`VRSQRTS` 命令は、(3 - op1 * op2) / 2 の計算を実行し、2 つの乗算と組み合わせることで、反復の 1 ステップを実行できます。この命令の機能は、次の擬似コード関数で定義されます。

```
// FPRSqrtStep()
// =============

bits(32) FPRSqrtStep(bits(32) op1, bits(32) op2)
    (type1,sign1,value1) = FPUnpack(op1, StandardFPSCRValue());
    (type2,sign2,value2) = FPUnpack(op2, StandardFPSCRValue());
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, StandardFPSCRValue());
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', 32);
        else
            product = FPMul(op1, op2, FALSE);
        result = FPDiv(FPSub(FPThree(32), product, FALSE), FPTwo(32), FALSE);
    return result;
```

入力値が範囲外の場合の結果を、表 A2-12 に示します。

表 A2-12 範囲外の入力に対する VRSQRTS の結果

入力、Vn[i]	入力、Vm[i]	結果、Vd[i]
すべての NaN	–	デフォルト NaN
–	すべての NaN	デフォルト NaN
+/- 0.0 または非正規化数	+/- 無限大	1.5
+/- 無限大	+/- 0.0 または非正規化数	1.5

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Conversions

The following functions perform conversions between half-precision and single-precision floating-point numbers.

```
// FPHalfToSingle()
// ================

bits(32) FPHalfToSingle(bits(16) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type,sign,value) = FPUnpack(operand, fpscr_val);
    if type == FPType_SNaN || type == FPType_QNaN then
        if fpscr_val<25> == '1' then // DN bit set
            result = FPDefaultNaN(32);
        else
            result = sign : '11111111 1' : operand<8:0> : Zeros(13);
        if type == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif type = FPType_Infinity then
        result = FPInfinity(sign, 32);
    elsif type = FPType_Zero then
        result = FPZero(sign, 32);
    else
        result = FPRound(value, 32, fpscr_val); // Rounding will be exact
    return result;

// FPSingleToHalf()
// ================

bits(16) FPSingleToHalf(bits(32) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type,sign,value) = FPUnpack(operand, fpscr_val);
    if type == FPType_SNaN || type == FPType_QNaN then
        if fpscr_val<26> == '1' then     // AH bit set
            result = FPZero(sign, 16);
        elsif fpscr_val<25> == '1' then  // DN bit set
            result = FPDefaultNaN(16);
        else
            result = sign : '11111 1' : operand<21:13>;
        if type == FPType_SNaN || fpscr_val<26> == '1' then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif type = FPType_Infinity then
        if fpscr_val<26> == '1' then // AH bit set
            result = sign : Ones(15);
            FPProcessException(FPExc_InvalidOp, fpscr_val);
        else
            result = FPInfinity(sign, 16);
    elsif type = FPType_Zero then
        result = FPZero(sign, 16);
    else
        result = FPRound(value, 16, fpscr_val);
    return result;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

変換

次の関数は、半精度と単精度の浮動小数点数の相互変換を実行します。

```
// FPHalfToSingle()
// ================

bits(32) FPHalfToSingle(bits(16) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type,sign,value) = FPUnpack(operand, fpscr_val);
    if type == FPType_SNaN || type == FPType_QNaN then
        if fpscr_val<25> == '1' then // DN bit set
            result = FPDefaultNaN(32);
        else
            result = sign : '11111111 1' : operand<8:0> : Zeros(13);
        if type == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif type = FPType_Infinity then
        result = FPInfinity(sign, 32);
    elsif type = FPType_Zero then
        result = FPZero(sign, 32);
    else
        result = FPRound(value, 32, fpscr_val); // Rounding will be exact
    return result;

// FPSingleToHalf()
// ================

bits(16) FPSingleToHalf(bits(32) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type,sign,value) = FPUnpack(operand, fpscr_val);
    if type == FPType_SNaN || type == FPType_QNaN then
        if fpscr_val<26> == '1' then     // AH bit set
            result = FPZero(sign, 16);
        elsif fpscr_val<25> == '1' then  // DN bit set
            result = FPDefaultNaN(16);
        else
            result = sign : '11111 1' : operand<21:13>;
        if type == FPType_SNaN || fpscr_val<26> == '1' then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif type = FPType_Infinity then
        if fpscr_val<26> == '1' then // AH bit set
            result = sign : Ones(15);
            FPProcessException(FPExc_InvalidOp, fpscr_val);
        else
            result = FPInfinity(sign, 16);
    elsif type = FPType_Zero then
        result = FPZero(sign, 16);
    else
        result = FPRound(value, 16, fpscr_val);
    return result;
```

次の関数は、単精度と倍精度の浮動小数点数の相互変換を実行します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The following functions perform conversions between single-precision and double-precision floating-point numbers.

```
// FPSingleToDouble()
// ==================

bits(64) FPSingleToDouble(bits(32) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type,sign,value) = FPUnpack(operand, fpscr_val);
    if type == FPType_SNaN || type == FPType_QNaN then
        if fpscr_val<25> == '1' then  // DN bit set
            result = FPDefaultNaN(64);
        else
            result = sign : '11111111111 1' : operand<21:0> : Zeros(29);
        if type == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif type = FPType_Infinity then
        result = FPInfinity(sign, 64);
    elsif type = FPType_Zero then
        result = FPZero(sign, 64);
    else
        result = FPRound(value, 64, fpscr_val);  // Rounding will be exact
    return result;

// FPDoubleToSingle()
// ==================

bits(32) FPDoubleToSingle(bits(64) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type,sign,value) = FPUnpack(operand, fpscr_val);
    if type == FPType_SNaN || type == FPType_QNaN then
        if fpscr_val<25> == '1' then  // DN bit set
            result = FPDefaultNaN(32);
        else
            result = sign : '11111111 1' : operand<50:29>;
        if type == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif type = FPType_Infinity then
        result = FPInfinity(sign, 32);
    elsif type = FPType_Zero then
        result = FPZero(sign, 32);
    else
        result = FPRound(value, 32, fpscr_val);
    return result;
```

The following functions perform conversions between floating-point numbers and integers or fixed-point numbers:

```
// FPToFixed()
// ===========

bits(M) FPToFixed(bits(N) operand, integer M, integer fraction_bits, boolean unsigned,
                  boolean round_towards_zero, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
// FPSingleToDouble()
// ==================

bits(64) FPSingleToDouble(bits(32) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type,sign,value) = FPUnpack(operand, fpscr_val);
    if type == FPType_SNaN || type == FPType_QNaN then
        if fpscr_val<25> == '1' then  // DN bit set
            result = FPDefaultNaN(64);
        else
            result = sign : '11111111111 1' : operand<21:0> : Zeros(29);
        if type == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif type = FPType_Infinity then
        result = FPInfinity(sign, 64);
    elsif type = FPType_Zero then
        result = FPZero(sign, 64);
    else
        result = FPRound(value, 64, fpscr_val);  // Rounding will be exact
    return result;

// FPDoubleToSingle()
// ==================

bits(32) FPDoubleToSingle(bits(64) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type,sign,value) = FPUnpack(operand, fpscr_val);
    if type == FPType_SNaN || type == FPType_QNaN then
        if fpscr_val<25> == '1' then  // DN bit set
            result = FPDefaultNaN(32);
        else
            result = sign : '11111111 1' : operand<50:29>;
        if type == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif type = FPType_Infinity then
        result = FPInfinity(sign, 32);
    elsif type = FPType_Zero then
        result = FPZero(sign, 32);
    else
        result = FPRound(value, 32, fpscr_val);
    return result;
```


次の関数は、浮動小数点数と整数または固定小数点数との相互変換を実行します。

```
// FPToFixed()
// ===========

bits(M) FPToFixed(bits(N) operand, integer M, integer fraction_bits, boolean unsigned,
                  boolean round_towards_zero, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    if round_towards_zero then fpscr_val<23:22> = '11';
    (type,sign,value) = FPUnpack(operand, fpscr_val);

    // For NaNs and infinities, FPUnpack() has produced a value that will round to the
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    if round_towards_zero then fpscr_val<23:22> = '11';
    (type,sign,value) = FPUnpack(operand, fpscr_val);

    // For NaNs and infinities, FPUnpack() has produced a value that will round to the
    // required result of the conversion. Also, the value produced for infinities will
    // cause the conversion to overflow and signal an Invalid Operation floating-point
    // exception as required. NaNs must also generate such a floating-point exception.
    if type == FPType_SNaN || type == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpscr_val);

    // Scale value by specified number of fraction bits, then start rounding to an integer
    // and determine the rounding error.
    value = value * 2^fraction_bits;
    int_result = RoundDown(value);
    error = value - int_result;

    // Apply the specified rounding mode.
    case fpscr_val<23:22> of
        when '00'  // Round to Nearest (rounding to even if exactly halfway)
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when '01'  // Round towards Plus Infinity
            round_up = (error != 0.0);
        when '10'  // Round towards Minus Infinity
            round_up = FALSE;
        when '11'  // Round towards Zero
            round_up = (error != 0.0 && int_result < 0);
    if round_up then int_result = int_result + 1;

    // Bitstring result is the integer result saturated to the destination size, with
    // saturation indicating overflow of the conversion (signaled as an Invalid
    // Operation floating-point exception).
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif error != 0 then
        FPProcessException(FPExc_Inexact, fpscr_val);

    return result;

// FixedToFP()
// ===========

bits(N) FixedToFP(bits(M) operand, integer N, integer fraction_bits, boolean unsigned,
                  boolean round_to_nearest, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    if round_to_nearest then fpscr_val<23:22> = '00';
    int_operand = if unsigned then UInt(operand) else SInt(operand);
    real_operand = int_operand / 2^fraction_bits;
    if real_operand == 0.0 then
        result = FPZero('0', N);
    else
        result = FPRound(real_operand, N, fpscr_val);
    return result;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

```
    // required result of the conversion. Also, the value produced for infinities will
    // cause the conversion to overflow and signal an Invalid Operation floating-point
    // exception as required. NaNs must also generate such a floating-point exception.
    if type == FPType_SNaN || type == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpscr_val);

    // Scale value by specified number of fraction bits, then start rounding to an integer
    // and determine the rounding error.
    value = value * 2^fraction_bits;
    int_result = RoundDown(value);
    error = value - int_result;

    // Apply the specified rounding mode.
    case fpscr_val<23:22> of
        when '00'  // Round to Nearest (rounding to even if exactly halfway)
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when '01'  // Round towards Plus Infinity
            round_up = (error != 0.0);
        when '10'  // Round towards Minus Infinity
            round_up = FALSE;
        when '11'  // Round towards Zero
            round_up = (error != 0.0 && int_result < 0);
    if round_up then int_result = int_result + 1;

    // Bitstring result is the integer result saturated to the destination size, with
    // saturation indicating overflow of the conversion (signaled as an Invalid
    // Operation floating-point exception).
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif error != 0 then
        FPProcessException(FPExc_Inexact, fpscr_val);

    return result;

// FixedToFP()
// ===========

bits(N) FixedToFP(bits(M) operand, integer N, integer fraction_bits, boolean unsigned,
                  boolean round_to_nearest, boolean fpscr_controlled)
    assert N == 32 || N == 64;
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    if round_to_nearest then fpscr_val<23:22> = '00';
    int_operand = if unsigned then UInt(operand) else SInt(operand);
    real_operand = int_operand / 2^fraction_bits;
    if real_operand == 0.0 then
        result = FPZero('0', N);
    else
        result = FPRound(real_operand, N, fpscr_val);
    return result;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.8 Polynomial arithmetic over {0,1}

The polynomial data type represents a polynomial in x of the form $b_{n-1}x^{n-1} + \ldots + b_1x + b_0$ where b_k is bit [k] of the value.

The coefficients 0 and 1 are manipulated using the rules of Boolean arithmetic:

- 0 + 0 = 1 + 1 = 0
- 0 + 1 = 1 + 0 = 1
- 0 * 0 = 0 * 1 = 1 * 0 = 0
- 1 * 1 = 1.

That is:

- adding two polynomials over {0,1} is the same as a bitwise exclusive OR
- multiplying two polynomials over {0,1} is the same as integer multiplication except that partial products are exclusive-ORed instead of being added.

A2.8.1 Pseudocode details of polynomial multiplication

In pseudocode, polynomial addition is described by the EOR operation on bitstrings.

Polynomial multiplication is described by the `PolynomialMult()` function:

```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = Zeros(M) : op2;
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.8 {0,1} における多項式算術演算

多項式のデータ型は、$b_{n-1}x^{n-1} + \cdots + b_1x + b_0$ の形の x の多項式を表現します。ここで、b_k は値のビット [k] です。

係数 0 と 1 は、ブール算術演算の規則を使用して操作されます。

- 0 + 0 = 1 + 1 = 0
- 0 + 1 = 1 + 0 = 1
- 0 ∗ 0 = 0 ∗ 1 = 1 ∗ 0 = 0
- 1 ∗ 1 = 1

これは、次のことを意味します。

- {0,1} における 2 つの多項式の加算は、ビット単位の排他的論理和と同じです。
- {0,1} における 2 つの多項式の乗算は、整数乗算と同じですが、部分積は加算ではなく排他的論理和となります。

A2.8.1 多項式乗算の擬似コードの詳細

擬似コードでは、多項式加算はビットストリングに対する排他的論理和として記述されます。

多項式の乗算は、`PolynomialMult()` 関数によって記述されます。


```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = Zeros(M) : op2;
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.9 Coprocessor support

Coprocessor space is used to extend the functionality of an ARM processor. There are sixteen coprocessors defined in the coprocessor instruction space. These are commonly known as CP0 to CP15. The following coprocessors are reserved by ARM for specific purposes:

- Coprocessor 15 (CP15) provides system control functionality. This includes architecture and feature identification, as well as control, status information and configuration support. The following sections describe CP15:
  - *CP15 registers for a VMSA implementation* on page B3-64
  - *CP15 registers for a PMSA implementation* on page B4-22.

  CP15 also provides performance monitor registers, see Chapter C9 *Performance Monitors*.
- Coprocessor 14 (CP14) supports:
  - debug, see Chapter C6 *Debug Register Interfaces*
  - the execution environment features defined by the architecture, see *Execution environment support* on page A2-69.
- Coprocessor 11 (CP11) supports double-precision floating-point operations.
- Coprocessor 10 (CP10) supports single-precision floating-point operations and the control and configuration of both the VFP and the Advanced SIMD architecture extensions.
- Coprocessors 8, 9, 12, and 13 are reserved for future use by ARM.

Note

Any implementation that includes either or both of the Advanced SIMD extension and the VFP extension must enable access to both CP10 and CP11, see *Enabling Advanced SIMD and floating-point support* on page B1-64.

In general, privileged access is required for:
- system control through CP15
- debug control and configuration
- access to the identification registers
- access to any register bits that enable or disable coprocessor features.

For details of the exact split between the privileged and unprivileged coprocessor operations see the relevant sections of this manual.

All load, store, branch and data operation instructions associated with floating-point, Advanced SIMD and execution environment support can execute unprivileged.

Coprocessors 0 to 7 can be used to provide vendor specific features.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A2.9 コプロセッサのサポート

コプロセッサ空間は、ARM プロセッサの機能拡張のために使用されます。コプロセッサ命令空間には、16 個のコプロセッサが定義されています。これらは一般に CP0 ～ CP15 と呼ばれます。次に示すコプロセッサは、ARM によって特定の目的のため予約されています。

- コプロセッサ 15 (CP15) は、システム制御機能を提供します。これには、アーキテクチャと機能の識別のほか、制御、ステータス情報、構成サポートが含まれます。CP15 の説明については、次のセクションを参照して下さい。
  - *VMSA 実装の CP15 レジスタ* P. B3-64
  - *PMSA 実装の CP15 レジスタ* P. B4-22

  CP15 は、パフォーマンスモニタ レジスタも提供します。第 C9 章「*パフォーマンスモニタ*」を参照して下さい。
- コプロセッサ 14 (CP14) は次の機能をサポートします。
  - デバッグ（第 C6 章「*デバッグレジスタ インタフェース*」を参照）
  - アーキテクチャによって定義される実行環境機能（*実行環境のサポート* P. A2-69 を参照）
- コプロセッサ 11 (CP11) は、倍精度の浮動小数点演算をサポートします。
- コプロセッサ 10 (CP10) は、単精度の浮動小数点演算に加え、VFP、アドバンスト SIMD の両方のアーキテクチャ拡張機能の制御と構成をサポートします。
- コプロセッサ 8、9、12、13 は、将来的な使用に備えて ARM により予約されています。

注

アドバンスト SIMD 拡張機能と VFP 拡張機能のいずれかまたは両方を含む実装はすべて、CP10 と CP11 へのアクセスを可能にする必要があります。*アドバンスト SIMD と浮動小数点のサポートの許可* P. B1-64 を参照して下さい。

一般に、次の操作には特権アクセスが必要です。
- CP15 を介したシステム制御
- デバッグ制御および構成
- 識別レジスタへのアクセス
- コプロセッサ機能を稼動または非稼動にする、すべてのレジスタビットへのアクセス

コプロセッサの特権動作と非特権動作との正確な区別の詳細については、本書の関連セクションを参照して下さい。

浮動小数点、アドバンスト SIMD、実行環境のサポートに関連するロード、ストア、分岐、データ処理命令はすべて、特権なしで実行可能です。

コプロセッサの 0 ～ 7 は、ベンダ固有の機能を提供するために使用可能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.10 Execution environment support

The Jazelle and ThumbEE states, introduced in *ISETSTATE* on page A2-15, support execution environments:

- The ThumbEE state is more generic, supporting a variant of the Thumb instruction set that minimizes the code size overhead generated by a *Just-In-Time* (JIT) or *Ahead-Of-Time* (AOT) compiler. JIT and AOT compilers convert execution environment source code to a native executable. For more information, see *Thumb Execution Environment*.
- The Jazelle state is specific to hardware acceleration of Java bytecodes. For more information, see *Jazelle direct bytecode execution support* on page A2-73.

A2.10.1 Thumb Execution Environment

Thumb Execution Environment (ThumbEE) is a variant of the Thumb instruction set designed as a target for dynamically generated code. This is code that is compiled on the device, from a portable bytecode or other intermediate or native representation, either shortly before or during execution. ThumbEE provides support for *Just-In-Time* (JIT), *Dynamic Adaptive Compilation* (DAC) and *Ahead-Of-Time* (AOT) compilers, but cannot interwork freely with the ARM and Thumb instruction sets.

ThumbEE is particularly suited to languages that feature managed pointers and array types.

ThumbEE executes instructions in the ThumbEE instruction set state. For information about instruction set states see *ISETSTATE* on page A2-15.

See *Thumb Execution Environment* on page B1-73 for system level information about ThumbEE.

ThumbEE instructions

In ThumbEE state, the processor executes almost the same instruction set as in Thumb state. However some instructions behave differently, some are removed, and some ThumbEE instructions are added.

The key differences are:

- additional instructions to change instruction set in both Thumb state and ThumbEE state
- new ThumbEE instructions to branch to handlers
- null pointer checking on load/store instructions executed in ThumbEE state
- an additional instruction in ThumbEE state to check array bounds
- some other modifications to load, store, and control flow instructions.

For more information about the ThumbEE instructions see Chapter A9 *ThumbEE*.

ThumbEE configuration

ThumbEE introduces two new registers:

- ThumbEE Configuration Register, TEECR. This contains a single bit, the ThumbEE configuration control bit, XED.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.10 実行環境のサポート

ISETSTATE P. A2-15 に紹介されている Jazelle 状態と ThumbEE 状態は、実行環境をサポートします。

- ThumbEE 状態のほうが汎用的で、Thumb 命令セットのバリアントをサポートし、*Just-In-Time* (JIT) または *Ahead-Of-Time* (AOT) コンパイラによって生成されるコードサイズのオーバヘッドを最小化します。JIT および AOT コンパイラは、実行環境のソースコードをネイティブの実行可能コードに変換します。詳細については、*Thumb 実行環境* を参照して下さい。
- Jazelle 状態は、Java バイトコードのハードウェアアクセラレーションに固有のものです。詳細については、*Jazelle 直接バイトコード実行のサポート* P. A2-73 を参照して下さい。

A2.10.1 Thumb 実行環境

Thumb 実行環境 (ThumbEE) は、動的に生成されるコードのターゲットとして設計された Thumb 命令セットのバリアントです。動的に生成されるコードとは、実行の直前または実行中に、移植可能なバイトコード、あるいは他の中間表現やネイティブ表現からデバイス上でコンパイルされるコードです。ThumbEE は、*Just-In-Time* (JIT)、*動的適応型コンパイル* (DAC)、*Ahead-Of-Time* (AOT) コンパイラをサポートしますが、ARM および Thumb 命令セットとの自由なインターワーキングは不可能です。

ThumbEE は、管理ポインタおよび管理配列型を持つ言語に特に適しています。

ThumbEE は、ThumbEE 命令セット状態で命令を実行します。命令セット状態の詳細については、*ISETSTATE* P. A2-15 を参照して下さい。

ThumbEE のシステムレベルの情報については、*Thumb 実行環境* P. B1-73 を参照して下さい。

ThumbEE 命令

ThumbEE 状態のプロセッサは、Thumb 状態とほとんど同じ命令セットを実行します。しかし、異なる動作をする命令や削除された命令もあり、新たに加えられた ThumbEE 命令もいくつかあります。

主な相違点は次のとおりです。

- Thumb 状態と ThumbEE 状態の両方で命令セットを変更するために追加された命令
- ハンドラに分岐する新しい ThumbEE 命令
- ThumbEE 状態で実行されるロード / ストア命令に対する NULL ポインタチェック機能
- ThumbEE 状態で配列の境界をチェックするための追加命令
- ロード、ストア、制御フロー命令に対する他のいくつかの変更

ThumbEE 命令の詳細については、第 A9 章「*ThumbEE*」を参照して下さい。

ThumbEE の構成

ThumbEE では、2 つの新しいレジスタが導入されています。

- ThumbEE 構成レジスタ、TEECR。このレジスタに含まれているビットは 1 つで、ThumbEE 構成制御ビット XED です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- ThumbEE Handler Base Register. This contains the base address for ThumbEE handlers.

  A handler is a short, commonly executed, sequence of instructions. It is typically, but not always, associated directly with one or more bytecodes or other intermediate language elements.

Changes to these CP14 registers have the same synchronization requirements as changes to the CP15 registers. These are described in:

- *Changes to CP15 registers and the memory order model* on page B3-77 for a VMSA implementation
- *Changes to CP15 registers and the memory order model* on page B4-28 for a PMSA implementation.

ThumbEE is an unprivileged, user-level facility, and there are no special provisions for using it securely. For more information, see *ThumbEE and the Security Extensions* on page B1-73.

ThumbEE Configuration Register (TEECR)

The ThumbEE Configuration Register (TEECR) controls unprivileged access to the ThumbEE Handler Base Register.

The TEECR is:

- a CP14 register
- a 32-bit register, with access rights that depend on the current privilege:
  - — the result of an unprivileged write to the register is UNDEFINED
  - — unprivileged reads, and privileged reads and writes, are permitted.
- when the Security Extensions are implemented, a Common register.

The format of the TEECR is:

31 … 1	0
UNK/SBZP	XED

Bits [31:1] UNK/SBZP.

XED, bit [0] Execution Environment Disable bit. Controls unprivileged access to the ThumbEE Handler Base Register:

0 Unprivileged access permitted.

1 Unprivileged access disabled.

The reset value of this bit is 0.

The effects of a write to this register on ThumbEE configuration are only guaranteed to be visible to subsequent instructions after the execution of an `ISB` instruction, an exception entry or an exception return. However, a read of this register always returns the value most recently written to the register.

To access the TEECR, read or write the CP14 registers with an `MRC` or `MCR` instruction with `<opc1>` set to 6, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC  p14, 6, <Rt>, c0, c0, 0   ; Read ThumbEE Configuration Register
MCR  p14, 6, <Rt>, c0, c0, 0   ; Write ThumbEE Configuration Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- ThumbEE ハンドラベース レジスタ。このレジスタには、ThumbEE ハンドラのベースアドレスが含まれます。

  ハンドラとは、短く、頻繁に実行される命令のシーケンスです。常にではありませんが、通常は、1 つ以上のバイトコードまたは他の中間言語要素に直接結びついています。

これらの CP14 レジスタの変更には、CP15 レジスタの変更と同様の同期要件が適用されます。説明については、次のセクションを参照して下さい。

- *CP15 レジスタの変更とメモリオーダ モデル* P. B3-77（VMSA 実装の場合）
- *CP15 レジスタの変更とメモリオーダ モデル* P. B4-28 （PMSA 実装の場合）

ThumbEE は、非特権でユーザレベルの機能であり、セキュアに使用するための特別な機構はありません。詳細については、*ThumbEE とセキュリティ拡張機能* P. B1-73 を参照して下さい。

ThumbEE 構成レジスタ (TEECR)

ThumbEE 構成レジスタ (TEECR) は、ThumbEE ハンドラベース レジスタへの非特権アクセスを制御します。

TEECR の特徴は次のとおりです。

- CP14 レジスタ
- 32 ビットレジスタで、アクセス権は現在の特権レベルに依存します。
  - ー レジスタへの非特権書き込みの結果は未定義です。
  - ー 非特権読み出し、および特権読み出しと書き込みが許可されます。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

TEECR のフォーマットは次のとおりです。

31 … 1	0
UNK/SBZP	XED

ビット [31:1] UNK/SBZP

XED、ビット [0] 実行環境ディセーブルビット。ThumbEE ハンドラ ベースレジスタへの非特権アクセスを制御します。

0 非特権アクセスが許可されます。

1 非特権アクセスは不可能です。

このビットのリセット値は 0 です。

ThumbEE 構成におけるこのレジスタへの書き込みの効果は、`ISB` 命令の実行、例外の開始、または例外からの復帰の後にある命令からのみ、認識可能であることが保証されています。しかし、このレジスタの読み出しは常に、最後にレジスタに書き込まれた値を返します。

TEECR にアクセスするには、`<opc1>` に 6、`<CRn>` に c0、`<CRm>` に c0、`<opc2>` に 0 を設定して、`MRC` または `MCR` 命令で CP14 レジスタを読み書きします。次に例を示します。

```
MRC  p14, 6, <Rt>, c0, c0, 0   ; Read ThumbEE Configuration Register
MCR  p14, 6, <Rt>, c0, c0, 0   ; Write ThumbEE Configuration Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ThumbEE Handler Base Register (TEEHBR)

The ThumbEE Handler Base Register (TEEHBR) holds the base address for ThumbEE handlers.

The TEEHBR is:

- a CP14 register
- a 32-bit read/write register, with access rights that depend on the current privilege and the value of the TEECR.XED bit:
  - — privileged accesses are always permitted
  - — when TEECR.XED == 0, unprivileged accesses are permitted
  - — when TEECR.XED == 1, the result of an unprivileged access is UNDEFINED.
- when the Security Extensions are implemented, a Common register.

The format of the TEEHBR is:

31	2 1 0
HandlerBase	SBZ

HandlerBase, bits [31:2]

The address of the ThumbEE Handler_00 implementation. This is the address of the first of the ThumbEE handlers.

The reset value of this field is UNKNOWN.

bits [1:0] Reserved, SBZ.

The effects of a write to this register on ThumbEE handler entry are only guaranteed to be visible to subsequent instructions after the execution of an ISB instruction, an exception entry or an exception return. However, a read of this register always returns the value most recently written to the register.

To access the TEEHBR, read or write the CP14 registers with an MRC or MCR instruction with <opc1> set to 6, <CRn> set to c1, <CRm> set to c0, and <opc2> set to 0. For example:

```
MRC  p14, 6, <Rt>, c1, c0, 0   ; Read ThumbEE Handler Base Register
MCR  p14, 6, <Rt>, c1, c0, 0   ; Write ThumbEE Handler Base Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ThumbEE ハンドラ ベースレジスタ (TEEHBR)

ThumbEE ハンドラ ベースレジスタ (TEEHBR) は、ThumbEE ハンドラのベースアドレスを保持します。

TEEHBR の特徴は次のとおりです。

- CP14 レジスタ
- 32 ビット読み出し / 書き込みレジスタで、アクセス権は現在の特権レベルと TEECR.XED ビットの値に依存します。
  - – 特権アクセスは常に許可されます。
  - – TEECR.XED = 0 の場合、非特権アクセスが許可されます。
  - – TEECR.XED = 1 の場合、非特権アクセスの結果は未定義です。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

TEEHBR のフォーマットは次のとおりです。

31	2 1 0
HandlerBase	SBZ

HandlerBase、ビット [31:2]

ThumbEE Handler_00 実装のアドレス。これは、最初の ThumbEE ハンドラのアドレスです。

このフィールドのリセット時の値は未知です。

ビット [1:0] 予約、SBZ

ThumbEE ハンドラ開始時におけるこのレジスタへの書き込みの効果は、ISB 命令の実行、例外の開始、または例外からの復帰の後にある命令からのみ、認識可能であることが保証されています。しかし、このレジスタの読み出しは常に、最後にレジスタに書き込まれた値を返します。

TEEHBR にアクセスするには、<opc1> に 6、<CRn> に c1、<CRm> に c0、<opc2> に 0 を設定して、MRC または MCR 命令で CP14 レジスタを読み書きします。次に例を示します。

```
MRC  p14, 6, <Rt>, c1, c0, 0   ; Read ThumbEE Handler Base Register
MCR  p14, 6, <Rt>, c1, c0, 0   ; Write ThumbEE Handler Base Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Use of HandlerBase

ThumbEE handlers are entered by reference to a HandlerBase address, defined by the TEEHBR. See *ThumbEE Handler Base Register (TEEHBR)* on page A2-71. Table A2-13 shows how the handlers are arranged in relation to the value of HandlerBase:

Table A2-13 Access to ThumbEE handlers

Offset from HandlerBase	Name	Value stored
-0x0008	IndexCheck	Branch to IndexCheck handler
-0x0004	NullCheck	Branch to NullCheck handler
+0x0000	Handler_00	Implementation of Handler_00
+0x0020	Handler_01	Implementation of Handler_01
...	...	...
+(0x0000 + 32n)	Handler_<n>	Implementation of Handler_<n>
...	...	Implementation of additional handlers

The IndexCheck occurs when a CHKA instruction detects an index out of range. For more information, see *CHKA* on page A9-15.

The NullCheck occurs when any memory access instruction is executed with a value of 0 in the base register. For more information, see *Null checking* on page A9-3.

Note

Checks are similar to conditional branches, with the added property that they clear the IT bits when taken.

Other handlers are called using explicit handler call instructions. For details see the following sections:

- *HB, HBL* on page A9-16
- *HBLP* on page A9-17
- *HBP* on page A9-18.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

HandlerBase の使用

ThumbEE ハンドラは、TEEHBR に定義されている HandlerBase アドレスへの参照によって開始されます。*ThumbEE ハンドラ ベースレジスタ (TEEHBR)* P. A2-71 を参照して下さい。
HandlerBase の値に対してハンドラがどのように配置されるかを、表 A2-13 に示します。

表 A2-13 ThumbEE ハンドラへのアクセス

HandlerBase からのオフセット	名前	ストアされる値
−0x0008	IndexCheck	IndexCheck ハンドラへの分岐
−0x0004	NullCheck	NullCheck ハンドラへの分岐
+0x0000	Handler_00	Handler_00 の実装
+0x0020	Handler_01	Handler_01 の実装
...	...	...
+(0x0000 + 32n)	Handler_〈n〉	Handler_〈n〉 の実装
...	...	その他のハンドラの実装

CHKA 命令で範囲外のインデクスが検出されると、IndexCheck が発生します。詳細については、*CHKA* P. A9-15 を参照して下さい。

ベースレジスタの値 0 でメモリアクセス命令が実行されると、NullCheck が発生します。詳細については、*NULL チェック* P. A9-3 を参照して下さい。

注

Checks は、条件分岐と似ていますが、取得時に IT ビットがクリアされるという特徴があります。

その他のハンドラは、明示的なハンドラ呼び出し命令で呼び出されます。詳細については、次のセクションを参照して下さい。

- *HB*、*HBL* P. A9-16
- *HBLP* P. A9-17
- *HBP* P. A9-18

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.10.2 Jazelle direct bytecode execution support

From ARMv5TEJ, the architecture requires every system to include an implementation of the Jazelle extension. The Jazelle extension provides architectural support for hardware acceleration of bytecode execution by a *Java Virtual Machine* (JVM).

In the simplest implementations of the Jazelle extension, the processor does not accelerate the execution of any bytecodes, and the JVM uses software routines to execute all bytecodes. Such an implementation is called a *trivial implementation* of the Jazelle extension, and has minimal additional cost compared with not implementing the Jazelle extension at all. An implementation that provides hardware acceleration of bytecode execution is a non-trivial Jazelle implementation.

These requirements for the Jazelle extension mean a JVM can be written to both:

- function correctly on all processors that include a Jazelle extension implementation
- automatically take advantage of the accelerated bytecode execution provided by a processor that includes a non-trivial implementation.

Typically, a non-trivial implementation of the Jazelle extension implements a subset of the bytecodes in hardware, choosing bytecodes that:

- can have simple hardware implementations
- account for a large percentage of bytecode execution time.

The required features of a non-trivial implementation are:

- provision of the Jazelle state
- a new instruction, `BXJ`, to enter Jazelle state
- system support that enables an operating system to regulate the use of the Jazelle extension hardware
- system support that enables a JVM to configure the Jazelle extension hardware to its specific needs.

The required features of a trivial implementation are:

- Normally, the Jazelle instruction set state is never entered. If an incorrect exception return causes entry to the Jazelle instruction set state, the next instruction executed is treated as UNDEFINED.
- The `BXJ` instruction behaves as a `BX` instruction.
- Configuration support that maintains the interface to the Jazelle extension is permanently disabled.

For more information about trivial implementations see *Trivial implementation of the Jazelle extension* on page B1-81.

A JVM that has been written to take advantage automatically of hardware-accelerated bytecode execution is known as an *Enabled JVM* (EJVM).

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.10.2 Jazelle 直接バイトコード実行のサポート

ARMv5TEJ およびそれ以降のアーキテクチャでは、すべてのシステムに Jazelle 拡張機能の実装が必須です。Jazelle 拡張機能は、*Java 仮想マシン* (JVM) によるバイトコード実行のハードウェアアクセラレーションを行うためのアーキテクチャサポートを提供します。

Jazelle 拡張機能の最も単純な実装では、プロセッサがバイトコードの実行を高速化するのではなく、JVM がソフトウェアルーチンを使用してすべてのバイトコードを実行します。このような実装は、Jazelle 拡張機能の*トリビアル実装*と呼ばれ、Jazelle 拡張機能を全く実装しない場合に比べ、ほとんどコストが増加しません。バイトコード実行のハードウェアアクセラレーションを提供する実装は、トリビアルではありません。

Jazelle 拡張機能のこのような条件は、次の両方の条件を満たすように JVM を記述できることを意味します。

- Jazelle 拡張機能の実装を含むすべてのプロセッサ上で正常に機能する。
- トリビアルでない実装を含むプロセッサで提供される、バイトコードの高速実行機能を自動的に利用する。

一般に、Jazelle 拡張機能のトリビアルでない実装では、バイトコード全体から次のようなサブセットを選択し、ハードウェアで実装します。

- 簡単なハードウェアで実装できる
- バイトコード実行時間の大部分を占める

非トリビアル実装の必要条件は、次のとおりです。

- Jazelle 状態に対応している
- Jazelle 状態を開始する新しい `BXJ` 命令
- オペレーティングシステムが Jazelle 拡張機能ハードウェアの使用を調整できるようにするシステムサポート
- JVM が特定のニーズに応じて Jazelle 拡張機能ハードウェアを設定できるようにするシステムサポート

トリビアル実装の必要条件は、次のとおりです。

- 通常、Jazelle 命令セット状態を開始することはありません。不正な例外復帰によって Jazelle 命令セット状態が開始された場合は、次に実行される命令が未定義として扱われます。
- `BXJ` 命令は `BX` 命令として動作します。
- Jazelle 拡張機能へのインタフェースを維持する構成サポートが恒久的に不可能にされています。

トリビアル実装の詳細については、*Jazelle 拡張機能のトリビアル実装* P. B1-81 を参照して下さい。

自動的にバイトコードをハードウェアで高速実行するよう記述された JVM は、*イネーブルド JVM*(EJVM) と呼ばれます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Subarchitectures

A processor implementation that includes the Jazelle extension expects the general-purpose register values and other resources of the ARM processor to conform to an interface standard defined by the Jazelle implementation when Jazelle state is entered and exited. For example, a specific general-purpose register might be reserved for use as the pointer to the current bytecode.

In order for an EJVM and associated debug support to function correctly, it must be written to comply with the interface standard defined by the acceleration hardware at Jazelle state execution entry and exit points.

An implementation of the Jazelle extension might define other configuration registers in addition to the architecturally defined ones.

The interface standard and any additional configuration registers used to communicate with the Jazelle extension are known collectively as the *subarchitecture* of the implementation. They are not described in this manual. Only EJVM implementations and debug or similar software can depend on the subarchitecture. All other software must rely only on the architectural definition of the Jazelle extension given in this manual. A particular subarchitecture is identified by reading the JIDR described in *Jazelle ID Register (JIDR)* on page A2-76.

Jazelle state

While the processor is in Jazelle state, it executes bytecode programs. A bytecode program is defined as an executable object that comprises one or more `class` files, or is derived from and functionally equivalent to one or more `class` files. See Lindholm and Yellin, *The Java Virtual Machine Specification 2nd Edition* for the definition of `class` files.

While the processor is in Jazelle state, the PC identifies the next JVM bytecode to be executed. A JVM bytecode is a bytecode defined in Lindholm and Yellin, or a functionally equivalent transformed version of a bytecode defined in Lindholm and Yellin.

For the Jazelle extension, the functionality of *Native methods*, as described in Lindholm and Yellin, must be specified using only instructions from the ARM, Thumb, and ThumbEE instruction sets.

An implementation of the Jazelle extension must not be documented or promoted as performing any task while it is in Jazelle state other than the acceleration of bytecode programs in accordance with this section and *The Java Virtual Machine Specification*.

Jazelle state entry instruction, `BXJ`

ARMv7 includes an ARM instruction similar to `BX`. The `BXJ` instruction has a single register operand that specifies a target instruction set state, ARM state or Thumb state, and branch target address for use if entry to Jazelle state is not available. For more information, see *BXJ* on page A8-64.

Correct entry into Jazelle state involves the EJVM executing the `BXJ` instruction at a time when both:

- the Jazelle extension Control and Configuration registers are initialized correctly, see *Application level configuration and control of the Jazelle extension* on page A2-75

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

サブアーキテクチャ

Jazelle 拡張機能を含むプロセッサ実装は、Jazelle 状態を開始および終了する際、汎用レジスタの値および ARM プロセッサの他のリソースが、Jazelle 実装によって定義されるインタフェース規格に適合していると想定しています。例えば、特定の汎用レジスタは、現在のバイトコードへのポインタとして使用するため予約されている可能性があります。

EJVM および関連するデバッグサポートが正常に機能するには、Jazelle 状態での実行開始および終了時点において、アクセラレーションハードウェアにより定義されるインタフェース規格に適合するよう記述されている必要があります。

Jazelle 拡張機能の実装は、アーキテクチャで定義されているものに加え、他のコンフィギュレーションレジスタも定義することがあります。

インタフェース規格、および Jazelle 拡張機能と通信するために使用される他のコンフィギュレーションレジスタをまとめて、実装の *サブアーキテクチャ*と呼びます。本書では、これらについては説明しません。EJVM 実装およびデバッグ、または同様のソフトウェアだけが、サブアーキテクチャに依存することができます。他のソフトウェアはすべて、本書で説明する Jazelle 拡張機能のアーキテクチャ定義にのみ依存できます。具体的なサブアーキテクチャは、*JazelleID レジスタ (JIDR)* P. A2-76 で説明する JIDR を読み出すことで特定されます。

Jazelle 状態

プロセッサは、Jazelle 状態にある間、バイトコードプログラムを実行します。バイトコードプログラムは、1 つ以上のクラスファイルで構成される、または 1 つ以上のクラスファイルから派生しそれに等しい機能を持つ、実行可能オブジェクトと定義されます。クラスファイルの定義については、Lindholm 氏と Yellin 氏による『*The Java Virtual Machine Specification 2nd Edition*』を参照して下さい。

プロセッサが Jazelle 状態にある間、実行する次の JVM バイトコードを特定するのは PC です。JVM バイトコードとは、Lindholm 氏と Yellin 氏によって定義されたバイトコード、または Lindholm 氏と Yellin 氏によって定義されたバイトコードと機能的に等しい変形バージョンです。

Jazelle 拡張機能において、Lindholm 氏と Yellin 氏により記述されている *ネイティブメソッド*の機能は、ARM、Thumb、ThumbEE 命令セットの命令だけを使用して指定する必要があります。

Jazelle 拡張機能の実装で、このセクションおよび『*The Java Virtual Machine Specification*』に従い、Jazelle 状態においてバイトコードプログラムの高速実行以外のタスクを実行すると説明したり、宣伝したりすることはできません。

Jazelle 状態開始命令、`BXJ`

ARMv7 には、`BX` に似た ARM 命令が含まれています。この `BXJ` 命令には、ターゲットの命令セット状態、すなわち ARM 状態または Thumb 状態を指定するとともに、Jazelle 状態の開始が実行できない場合に使用する分岐ターゲットアドレスを指定する、単一のレジスタオペランドがあります。詳細については、*BXJ* P. A8-64 を参照して下さい。

Jazelle 状態の正常な開始においては、次の条件の両方が満たされたとき、EJVM が `BXJ` 命令を実行します。

- Jazelle 拡張機能の制御レジスタと構成レジスタが正常に初期化された。*アプリケーションレベルの構成と Jazelle 拡張機能の制御* P. A2-75 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- application level registers and any additional configuration registers are initialized as required by the subarchitecture of the implementation.

Executing BXJ with Jazelle extension enabled

Executing a BXJ instruction when the JMCR.JE bit is 1, see *Jazelle Main Configuration Register (JMCR)* on page A2-77, causes the Jazelle hardware to do one of the following:

- enter Jazelle state and start executing bytecodes directly from a SUBARCHITECTURE DEFINED address
- branch to a SUBARCHITECTURE DEFINED handler.

Which of these occurs is SUBARCHITECTURE DEFINED.

The Jazelle subarchitecture can use Application Level registers (but not System Level registers) to transfer information between the Jazelle extension and the EJVM. There are SUBARCHITECTURE DEFINED restrictions on what Application Level registers must contain when a BXJ instruction is executed, and Application Level registers have SUBARCHITECTURE DEFINED values when Jazelle state execution ends and ARM or Thumb state execution resumes.

Jazelle subarchitectures and implementations must not use any unallocated bits in Application Level registers such as the CPSR or FPSCR. All such bits are reserved for future expansion of the ARM architecture.

Executing BXJ with Jazelle extension disabled

If a BXJ instruction is executed when the JMCR.JE bit is 0, it is executed identically to a BX instruction with the same register operand.

This means that BXJ instructions can be executed freely when the JMCR.JE bit is 0. In particular, if an EJVM determines that it is executing on a processor whose Jazelle extension implementation is trivial or uses an incompatible subarchitecture, it can set JE == 0 and execute correctly. In this case it executes without the benefit of any Jazelle hardware acceleration that might be present.

Application level configuration and control of the Jazelle extension

All registers associated with the Jazelle extension are implemented in coprocessor space as part of coprocessor 14 (CP14). The registers are accessed using the instructions:

- MCR, see *MCR, MCR2* on page A8-186
- MRC, see *MRC, MRC2* on page A8-202.

In a non-trivial implementation at least three registers are required. These are described in:

- *Jazelle ID Register (JIDR)* on page A2-76
- *Jazelle Main Configuration Register (JMCR)* on page A2-77
- *Jazelle OS Control Register (JOSCR)* on page B1-77.

Additional configuration registers might be provided and are SUBARCHITECTURE DEFINED.

The following rules apply to all Jazelle extension control and configuration registers:

- All configuration registers are accessed by CP14 MRC and MCR instructions with <opc1> set to 7.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- アプリケーションレベル レジスタおよび他の構成レジスタが、実装のサブアーキテクチャの要求に従って初期化された。

Jazelle 拡張機能が稼働中である場合の BXJ の実行

JMCR.JE ビットが 1 のとき（*Jazelle メイン構成レジスタ (JMCR)* P. A2-77 を参照）に BXJ 命令を実行すると、Jazelle ハードウェアは次のいずれかを実行します。

- Jazelle 状態を開始し、サブアーキテクチャ定義のアドレスから直接バイトコードの実行を開始する。
- サブアーキテクチャ定義のハンドラへ分岐する。

どちらの動作が行われるかは、サブアーキテクチャ定義です。

Jazelle サブアーキテクチャは、アプリケーションレベルのレジスタ（システムレベルのレジスタを除く）を使用して、Jazelle 拡張機能と EJVM との間で情報を転送できます。BXJ 命令を実行する際にアプリケーションレベル レジスタが保持しなければならないものについてはサブアーキテクチャ定義の制約があります。また、Jazelle 状態の実行が終了し、ARM または Thumb 状態の実行が再開される際、アプリケーション レベルのレジスタの値はサブアーキテクチャ定義です。

Jazelle のサブアーキテクチャと実装で、CPSR や FPSCR などのアプリケーションレベル レジスタに含まれている、未割り当てのビットを使用することはできません。これらのビットはすべて、ARM アーキテクチャの将来的な拡張のために予約されています。

Jazelle 拡張機能が非稼動な場合の BXJ の実行

JMCR.JE ビットが 0 の場合、BXJ 命令は BX 命令と同じレジスタオペランドを使用して、同様に実行されます。

つまり、JMCR.JE ビットが 0 の場合、BXJ 命令は自由に実行できます。特に、プロセッサの Jazelle 拡張機能の実装がトリビアルであるか、プロセッサが互換性のないサブアーキテクチャを使用していると EJVM が判断した場合、EJVM は JE = 0 にセットし、正常に実行できます。この場合は、実行に際して Jazelle ハードウェアアクセラレーションの利点が得られることはありません。

アプリケーションレベルの構成と Jazelle 拡張機能の制御

Jazelle 拡張機能に関連するすべてのレジスタは、コプロセッサ 14 (CP14) の一部としてコプロセッサ空間に実装されます。これらのレジスタにアクセスするには、次の命令を使用します。

- MCR。*MCR*、*MCR2* P. A8-186 を参照して下さい。
- MRC。*MRC*、*MRC2* P. A8-202 を参照して下さい。

非トリビアル実装では、少なくとも 3 つのレジスタが必要です。これらのレジスタの説明については、次のセクションを参照して下さい。

- *JazelleID レジスタ (JIDR)* P. A2-76
- *Jazelle メイン構成レジスタ (JMCR)* P. A2-77
- *Jazelle OS 制御レジスタ (JOSCR)* P. B1-77

追加の構成レジスタが提供される場合もあります。これらはサブアーキテクチャ定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- The values contained in configuration registers are changed only by the execution of MCR instructions. In particular, they are never changed by Jazelle state execution of bytecodes.
- The access policy for the required registers is fully defined in their descriptions. With unprivileged operation:
  — all MCR accesses to the JIDR are UNDEFINED
  — MRC and MCR accesses that are restricted to privileged modes are UNDEFINED.

  The access policy of other configuration registers is SUBARCHITECTURE DEFINED.
- When the Security Extensions are implemented, the registers are common to the Secure and Non-secure security states. For more information, see *Effect of the Security Extensions on the CP15 registers* on page B3-71. This section applies to some CP14 registers as well as to the CP15 registers.
- When a configuration register is readable, reading the register returns the last value written to it. Reading a readable configuration register has no side effects.

  When a configuration register is not readable, attempting to read it returns an UNKNOWN value.
- When a configuration register can be written, the effect of writing to it must be idempotent. That is, the overall effect of writing the same value more than once must not differ from the effect of writing it once.

Changes to these CP14 registers have the same synchronization requirements as changes to the CP15 registers. These are described in:

- *Changes to CP15 registers and the memory order model* on page B3-77 for a VMSA implementation
- *Changes to CP15 registers and the memory order model* on page B4-28 for a PMSA implementation.

For more information, see *Jazelle state configuration and control* on page B1-77.

Jazelle ID Register (JIDR)

The Jazelle ID Register (JIDR) enables an EJVM to determine the architecture and subarchitecture under which it is running.

The JIDR is:

- a CP14 register
- a 32-bit read-only register
- accessible during privileged and unprivileged execution
- when the Security Extensions are implemented, a Common register, see *Common CP15 registers* on page B3-74.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

すべての Jazelle 拡張機能制御および構成レジスタには、次の規則が適用されます。

- すべての構成レジスタは、CP14 の MRC 命令と MCR 命令で、<opc1> を 7 に設定してアクセスされます。
- 構成レジスタに格納される値は、MCR 命令の実行によってのみ変更されます。特に、Jazelle 状態のバイトコード実行によっては変更されません。
- 必要なレジスタに対するアクセスポリシーは、レジスタの説明に詳細に定義されています。非特権動作の場合、次のようになります。
  – JIDR に対するすべての MCR アクセスは未定義です。
  – 特権モードに限られる MRC アクセスと MCR アクセスは未定義です。

  他の構成レジスタのアクセスポリシーは、サブアーキテクチャ定義です。
- セキュリティ拡張機能が実装されている場合、レジスタはセキュアと非セキュアのセキュリティ状態で共通です。詳細については、*CP15 レジスタに対するセキュリティ拡張機能の影響* P. B3-71 を参照して下さい。このセクションでは、一部の CP14 レジスタと CP15 レジスタについて説明しています。
- 構成レジスタが読み出し可能なとき、レジスタの読み出しはレジスタに最後に書き込まれた値を返します。読み出し可能な構成レジスタの読み出しに副作用はありません。

  構成レジスタが読み出し可能でないとき、読み出そうとすると未知の値が返されます。
- 構成レジスタが書き込み可能なとき、書き込みの効果はベキ等の必要があります。つまり、同じ値を 2 度以上書き込む場合の全体的な効果が、1 度書き込む場合の効果と異なってはいけません。

これらの CP14 レジスタの変更には、CP15 レジスタの変更と同様の同期要件が適用されます。これらについては、次のセクションを参照して下さい。

- *CP15 レジスタの変更とメモリオーダ モデル* P. B3-77（VMSA 実装の場合）
- *CP15 レジスタの変更とメモリオーダ モデル* P. B4-28 （PMSA 実装の場合）

詳細については、*Jazelle 状態の構成と制御* P. B1-77 を参照して下さい。

JazelleID レジスタ (JIDR)

JazelleID レジスタ (JIDR) により、EJVM は実行されているアーキテクチャとサブアーキテクチャを判断できます。

JIDR の特徴は次のとおりです。

- CP14 レジスタ
- 32 ビットの読み出し専用レジスタ
- 特権実行時および非特権実行時にアクセス可能です。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。*共通の CP15 レジスタ* P. B3-74 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the JIDR is:

31–28	27–20	19–12	11–0
Architecture	Implementer	Subarchitecture	SUBARCHITECTURE DEFINED

Architecture, bits [31:28]

Architecture code. This uses the same Architecture code that appears in the Main ID register in coprocessor 15, see *c0, Main ID Register (MIDR)* on page B3-81 (VMSA implementation) or *c0, Main ID Register (MIDR)* on page B4-32 (PMSA implementation).

Implementer, bits [27:20]

Implementer code of the designer of the subarchitecture. This uses the same Implementer code that appears in the Main ID register in coprocessor 15, see *c0, Main ID Register (MIDR)* on page B3-81 (VMSA implementation) or *c0, Main ID Register (MIDR)* on page B4-32 (PMSA implementation).

If the trivial implementation of the Jazelle extension is used, the Implementer code is `0x00`.

Subarchitecture, bits [19:12]

Contain the subarchitecture code. The following subarchitecture code is defined:

`0x00` Jazelle v1 subarchitecture, or trivial implementation of Jazelle extension if Implementer code is `0x00`.

bits [11:0] Contain additional SUBARCHITECTURE DEFINED information.

To access the JIDR, read the CP14 registers with an `MRC` instruction with `<opc1>` set to 7, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC  p14, 7, <Rt>, c0, c0, 0   ; Read Jazelle ID register
```

Jazelle Main Configuration Register (JMCR)

The Jazelle Main Configuration Register (JMCR) controls the Jazelle extension.

The JMCR is:

- a CP14 register
- a 32-bit register, with access rights that depend on the current privilege:
  - for privileged operations the register is read/write
  - for unprivileged operations, the register is normally write-only
- when the Security Extensions are implemented, a Common register, see *Common CP15 registers* on page B3-74.

For more information about unprivileged access restrictions see *Access to Jazelle registers* on page A2-78.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

JIDR のフォーマットは次のとおりです。

31–28	27–20	19–12	11–0
アーキテクチャ	実装者	サブアーキテクチャ	サブアーキテクチャ定義

アーキテクチャ、ビット [31:28]

アーキテクチャコード。コプロセッサ 15 のメイン ID レジスタにあるのと同じアーキテクチャコードを使用します。*c0、メイン ID レジスタ (MIDR)* P. B3-81（VMSA 実装の場合）、または *c0、メイン ID レジスタ (MIDR)* P. B4-32（PMSA 実装の場合）を参照して下さい。

実装者、ビット [27:20]

サブアーキテクチャ設計者の実装者コード。コプロセッサ 15 のメイン ID レジスタにあるのと同じ実装者コードを使用します。*c0、メイン ID レジスタ (MIDR)* P. B3-81（VMSA 実装の場合）、または *c0、メイン ID レジスタ (MIDR)* P. B4-32（PMSA 実装の場合）を参照して下さい。

Jazelle 拡張機能のトリビアル実装が使用されている場合、実装者コードは `0x00` です。

サブアーキテクチャ、ビット [19:12]

サブアーキテクチャコードが含まれます。次のサブアーキテクチャコードが定義されています。

`0x00` Jazelle v1 サブアーキテクチャ、または実装者コードが `0x00` の場合は Jazelle 拡張機能のトリビアル実装です。

ビット [11:0] その他のサブアーキテクチャ定義の情報が含まれます。

JIDR にアクセスするには、`<opc1>` に 7、`<CRn>` に c0、`<CRm>` に c0、`<opc2>` に 0 を設定して、`MRC` 命令で CP14 レジスタを読み出します。次に例を示します。

```
MRC  p14, 7, <Rt>, c0, c0, 0   ; Read Jazelle ID register
```

Jazelle メイン構成レジスタ (JMCR)

Jazelle メイン構成レジスタ (JMCR) は、Jazelle 拡張機能を制御します。

JMCR の特徴は次のとおりです。

- CP14 レジスタ
- 32 ビットレジスタで、アクセス権は現在の特権レベルに依存します。
  - 特権動作の場合、レジスタは読み出し / 書き込み可能です。
  - 非特権動作の場合、レジスタは通常は書き込み専用です。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。*共通の CP15 レジスタ* P. B3-74 を参照して下さい。

非特権アクセス制限の詳細については、*Jazelle レジスタへのアクセス* P. A2-78 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the JMCR is:

31 … 1	0
SUBARCHITECTURE DEFINED	JE

bit [31:1] SUBARCHITECTURE DEFINED information.

JE, bit [0] Jazelle Enable bit:

0 Jazelle extension disabled. The BXJ instruction does not cause Jazelle state execution. BXJ behaves exactly as a BX instruction, see *Jazelle state entry instruction, BXJ* on page A2-74.

1 Jazelle extension enabled.

The reset value of this bit is 0.

To access the JMCR, read or write the CP14 registers with an MRC or MCR instruction with <opc1> set to 7, <CRn> set to c2, <CRm> set to c0, and <opc2> set to 0. For example:

```
MRC  p14, 7, <Rt>, c2, c0, 0   ; Read Jazelle Main Configuration register
MCR  p14, 7, <Rt>, c2, c0, 0   ; Write Jazelle Main Configuration register
```

Access to Jazelle registers

Table A2-14 shows the access permissions for the Jazelle registers, and how unprivileged access to the registers depends on the value of the JOSCR.

Table A2-14 Access to Jazelle registers

Jazelle register	Unprivileged access: JOSCR.CD == 0[a]	Unprivileged access: JOSCR.CD == 1[a]	Privileged access
JIDR	Read access permitted	Read and write access UNDEFINED	Read access permitted
	Write access ignored		Write access ignored
JMCR	Read access UNDEFINED	Read and write access UNDEFINED	Read and write access permitted
	Write access permitted		
SUBARCHITECTURE DEFINED configuration registers	Read access UNDEFINED	Read and write access UNDEFINED	Read access SUBARCHITECTURE DEFINED
	Write access permitted		Write access permitted

a. See *Jazelle OS Control Register (JOSCR)* on page B1-77.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

JMCR のフォーマットは次のとおりです。

31 … 1	0
サブアーキテクチャ定義	JE

ビット [31:1] サブアーキテクチャ定義の情報。

JE、ビット [0] Jazelle イネーブルビット。

0 Jazelle 拡張機能は非稼動です。BXJ 命令によって Jazelle 状態の実行が開始されません。BXJ 命令は、BX 命令と全く同じ動作をします。*Jazelle 状態開始命令、BXJ* P. A2-74 を参照して下さい。

1 Jazelle 拡張機能が稼動中です。

このビットのリセット時の値は 0 です。

JMCR にアクセスするには、<opc1> に 7、<CRn> に c2、<CRm> に c0、<opc2> に 0 を設定して、MRC または MCR 命令で CP14 レジスタを読み書きします。次に例を示します。

```
MRC  p14, 7, <Rt>, c2, c0, 0   ; Read Jazelle Main Configuration register
MCR  p14, 7, <Rt>, c2, c0, 0   ; Write Jazelle Main Configuration register
```

Jazelle レジスタへのアクセス

Jazelle レジスタのアクセス許可と、レジスタへの非特権アクセスが JOSCR の値にどのように依存するかを、表 A2-14 に示します。

表 A2-14 Jazelle レジスタへのアクセス

Jazelle レジスタ	非特権アクセス: JOSCR.CD ＝ 0[a]	非特権アクセス: JOSCR.CD ＝ 1[a]	特権アクセス
JIDR	読み出しアクセスを許可	読み出し / 書き込みアクセスは未定義	読み出しアクセスを許可
	書き込みアクセスを無視		書き込みアクセスを無視
JMCR	読み出しアクセスは未定義	読み出し / 書き込みアクセスは未定義	読み出し / 書き込みアクセスを許可
	書き込みアクセスを許可		
サブアーキテクチャ定義の構成レジスタ	読み出しアクセスは未定義	読み出し / 書き込みアクセスは未定義	読み出しアクセスはサブアーキテクチャ定義
	書き込みアクセスを許可		書き込みアクセスを許可

a. 詳細については、*Jazelle OS 制御レジスタ (JOSCR)* P. B1-77 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

EJVM operation

The following subsections summarize how an EJVM must operate, to meet the requirements of the architecture:

- *Initialization*
- *Bytecode execution*
- *Jazelle exception conditions*
- *Other considerations* on page A2-80.

Initialization

During initialization, the EJVM must first check which subarchitecture is present, by checking the Implementer and Subarchitecture codes in the value read from the JIDR.

If the EJVM is incompatible with the subarchitecture, it must do one of the following:

- write a value with JE == 0 to the JMCR
- if unaccelerated bytecode execution is unacceptable, generate an error.

If the EJVM is compatible with the subarchitecture, it must write its required configuration to the JMCR and any SUBARCHITECTURE DEFINED configuration registers.

Bytecode execution

The EJVM must contain a handler for each bytecode.

The EJVM initiates bytecode execution by executing a `BXJ` instruction with:

- the register operand specifying the target address of the bytecode handler for the first bytecode of the program
- the Application Level registers set up in accordance with the SUBARCHITECTURE DEFINED interface standard.

The bytecode handler:

- performs the data-processing operations required by the bytecode indicated
- determines the address of the next bytecode to be executed
- determines the address of the handler for that bytecode
- performs a `BXJ` to that handler address with the registers again set up to the SUBARCHITECTURE DEFINED interface standard.

Jazelle exception conditions

During bytecode execution, the EJVM might encounter SUBARCHITECTURE DEFINED Jazelle exception conditions that must be resolved by a software handler. For example, in the case of a *configuration invalid* handler, the handler rewrites the desired configuration to the JMCR and to any SUBARCHITECTURE DEFINED configuration registers.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

EJVM の動作

以下のサブセクションでは、アーキテクチャの条件を満たすために、EJVM がどのように動作しなければならないかの要約について説明します。

- *初期化*
- *バイトコードの実行*
- *Jazelle 例外条件*
- *その他の留意事項* P. A2- 80

初期化

初期化時、EJVM は最初に、JIDR から読み出した値に含まれる実装者コードとサブアーキテクチャコードをチェックし、どのようなサブアーキテクチャが存在するかを確認する必要があります。

EJVM がサブアーキテクチャと互換性を持たない場合、EJVM は次のいずれかの動作を行う必要があります。

- JE == 0 の値を JMCR に書き込む。
- 高速化されていないバイトコード実行が許容できない場合は、エラーを生成する。

EJVM がサブアーキテクチャと互換性を持つ場合、EJVM は、必要な構成を JMCR および他のサブアーキテクチャ定義の構成レジスタに書き込む必要があります。

バイトコードの実行

EJVM は、各バイトコードのハンドラを含む必要があります。

EJVM は、次のレジスタを使用し、`BXJ` 命令を実行して、バイトコードの実行を開始します。

- プログラムの最初のバイトコードについて、バイトコードハンドラのターゲットアドレスを指定するレジスタオペランド
- サブアーキテクチャ定義のインタフェース規格に従ってセットアップされたアプリケーションレベル レジスタ

バイトコードハンドラは、次の動作を行います。

- 指定されたバイトコードにより要求されるデータ処理動作を実行する。
- 次に実行されるバイトコードのアドレスを決定する。
- そのバイトコードのハンドラのアドレスを決定する。
- 同様にサブアーキテクチャ定義のインタフェース規格に従ってセットアップされたレジスタを使用して、そのハンドラアドレスへの `BXJ` を実行する。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

On entry to a Jazelle exception condition handler the contents of the Application Level registers are SUBARCHITECTURE DEFINED. This interface to the Jazelle exception condition handler might differ from the interface standard for the bytecode handler, in order to supply information about the Jazelle exception condition.

The Jazelle exception condition handler:

- resolves the Jazelle exception condition
- determines the address of the next bytecode to be executed
- determines the address of the handler for that bytecode
- performs a BXJ to that handler address with the registers again set up to the SUBARCHITECTURE DEFINED interface standard.

Other considerations

To ensure application execution and correct interaction with an operating system, an EJVM must only perform operations that are permitted in unprivileged operation. In particular, for register accesses they must only:

- read the JIDR,
- write to the JMCR, and other configuration registers.

An EJVM must not attempt to access the JOSCR.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

Jazelle 例外条件

バイトコードの実行中、EJVM は、ソフトウェアハンドラによって解決しなければならないサブアーキテクチャ定義の Jazelle 例外条件に遭遇することがあります。例えば、*構成無効*ハンドラの場合、必要な構成をハンドラが、JMCR およびサブアーキテクチャ定義の構成レジスタに再書き込みします。

Jazelle 例外条件ハンドラの開始時、アプリケーションレベル レジスタの内容はサブアーキテクチャ定義です。Jazelle 例外条件ハンドラへのこのインタフェースは、Jazelle 例外条件に関する情報を供給するため、バイトコードハンドラ用のインタフェース規格と異なる場合があります。

Jazelle 例外条件ハンドラは、次の動作を行います。

- Jazelle 例外条件を解決する。
- 次に実行されるバイトコードのアドレスを決定する。
- そのバイトコードのハンドラのアドレスを決定する。
- 同様にサブアーキテクチャ定義のインタフェース規格に従ってセットアップされたレジスタを使用して、そのハンドラアドレスへの BXJ を実行する。

その他の留意事項

アプリケーションの実行とオペレーティングシステムとの正しい相互作用を確実にするため、EJVM は非特権動作で許可されている処理だけを実行する必要があります。特にレジスタアクセスについては、次の動作しか行えません。

- JIDR の読み出し
- JMCR およびその他の構成レジスタへの書き込み

EJVM は、JOSCR へのアクセスを試みることはできません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.11 Exceptions, debug events and checks

ARMv7 uses the following terms to describe various types of exceptional condition:

Exceptions In the ARM architecture, *exceptions* cause entry into a privileged mode and execution of a software handler for the exception.

Note

The terms *floating-point exception* and *Jazelle exception condition* do not use this meaning of *exception*. These terms are described later in this list.

Exceptions include:

- reset
- interrupts
- memory system aborts
- undefined instructions
- supervisor calls (SVCs).

Most details of exception handling are not visible to application-level code, and are described in *Exceptions* on page B1-30. Aspects that are visible to application-level code are:

- The `SVC` instruction causes an SVC exception. This provides a mechanism for unprivileged code to make a call to the operating system (or other privileged component of the software system).
- If the Security Extensions are implemented, the `SMC` instruction causes an SMC exception, but only if it is executed in a privileged mode. Unprivileged code can only cause SMC exceptions to occur by methods defined by the operating system (or other privileged component of the software system).
- The `WFI` instruction provides a hint that nothing needs to be done until an interrupt or similar exception is taken, see *Wait For Interrupt* on page B1-47. This permits the processor to enter a low-power state until that happens.
- The `WFE` instruction provides a hint that nothing needs to be done until either an *event* is generated by an `SEV` instruction or an interrupt or similar exception is taken, see *Wait For Event and Send Event* on page B1-44. This permits the processor to enter a low-power state until one of these happens.
- The `YIELD` instruction provides a hint that the current execution thread is of low importance, see *The Yield instruction* on page A2-82.

Floating-point exceptions

These relate to exceptional conditions encountered during floating-point arithmetic, such as division by zero or overflow. For more information see:

- *Floating-point exceptions* on page A2-42
- *Floating-point Status and Control Register (FPSCR)* on page A2-28
- ANSI/IEEE Std. 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A2.11 例外、デバッグイベント、チェック

ARMv7 では、各種の例外条件について説明するため、次の用語が使用されます。

例外 ARM アーキテクチャにおいて、*例外*とは特権モードの開始と例外に対応するソフトウェアハンドラの実行を引き起こすものです。

注

*浮動小数点例外*と *Jazelle 例外条件*という用語は、*例外*のこの意味を使用しません。これらの用語については後で説明します。

例外には、次のものが含まれます。

- リセット
- 割り込み
- メモリシステム アボート
- 未定義命令
- スーパバイザコール (SVC)

例外処理の詳細のほとんどは、アプリケーションレベルのコードによって認識されません。これについては、*例外* P. B1-30 で説明します。アプリケーションレベルのコードが認識する側面は、次のとおりです。

- `SVC` 命令は、SVC 例外を引き起こします。これは、非特権コードがオペレーティングシステム（またはソフトウェアシステムに含まれる他の特権コンポーネント）を呼び出す機構を提供します。
- セキュリティ拡張機能が実装されている場合、`SMC` 命令は SMC 例外を引き起こしますが、これは特権モードで実行された場合のみです。非特権コードは、オペレーティングシステム（またはソフトウェアシステムに含まれる他の特権コンポーネント）に定義されている方法によってのみ、SMC 例外を発生させることができます。
- `WFI` 命令は、割り込みまたは同様の例外が取得されるまで、何もする必要がないことを示すヒントです。*割り込み待ち* P. B1-47 を参照して下さい。これにより、そのような例外が取得されるまでプロセッサは省電力状態に入ることができます。
- `WFE` 命令は、`SEV` 命令によって*イベント*が生成されるか、割り込みまたは同様の例外が取得されるまで、何もする必要がないことを示すヒントです。*イベント待ちとイベント送信* P. B1-44 を参照して下さい。これにより、その時点までプロセッサは省電力状態に入ることができます。
- `YIELD` 命令は、現在の実行スレッドの重要性が低いことを示すヒントです。*Yield 命令* P. A2-82 を参照して下さい。

浮動小数点例外

これらの例外は、0 による除算やオーバフローなど、浮動小数点算術演算で発生する例外条件に関連します。詳細については、次のものを参照して下さい。

- *浮動小数点例外* P. A2-42

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Jazelle exception conditions

These are conditions that cause Jazelle hardware acceleration to exit into a software handler, as described in *Jazelle exception conditions* on page A2-79.

Debug events These are conditions that cause a debug system to take action. Most aspects of debug events are not visible to application-level code, and are described in Chapter C3 *Debug Events*. Aspects that are visible to application-level code include:

- The `BKPT` instruction causes a BKPT Instruction debug event to occur, see *BKPT Instruction debug events* on page C3-20.
- The `DBG` instruction provides a hint to the debug system.

Checks These are provided in the ThumbEE extension. A check causes an unconditional branch to a specific handler entry point. The base address of the ThumbEE check handlers is held in the TEEHBR, see *ThumbEE Handler Base Register (TEEHBR)* on page A2-71.

A2.11.1 The Yield instruction

In a *Symmetric Multi-Threading* (SMT) design, a thread can use a Yield instruction to give a hint to the processor that it is running on. The Yield hint indicates that whatever the thread is currently doing is of low importance, and so could yield. For example, the thread might be sitting in a spin-lock. Similar behavior might be used to modify the arbitration priority of the snoop bus in a multiprocessor (MP) system. Defining such an instruction permits binary compatibility between SMT and SMP systems.

ARMv7 defines a `YIELD` instruction as a specific NOP-hint instruction, see *YIELD* on page A8-812.

The `YIELD` instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use on migration to a multiprocessor or multithreading system. Operating systems can use `YIELD` in places where a yield hint is wanted, knowing that it will be treated as a `NOP` if there is no implementation benefit.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- *浮動小数点ステータスおよび制御レジスタ (FPSCR)* P. A2-28
- ANSI/IEEE Std.754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*

Jazelle 例外条件

Jazelle ハードウェアアクセラレーションが終了してソフトウェアハンドラが開始する条件です。*Jazelle 例外条件* P. A2-80 を参照して下さい。

デバッグイベント

デバッグシステムが動作を開始する条件です。デバッグイベントのほとんどの側面は、アプリケーションレベルのコードによって認識されません。これについては、第 C3 章「*デバッグイベント*」で説明します。アプリケーションレベルのコードが認識する側面は、次のとおりです。

- `BKPT` 命令は、BKPT 命令デバッグイベントを引き起こします。*BKPT 命令デバッグイベント* P. C3-20 を参照して下さい。
- `DBG` 命令は、デバッグシステムにヒントを提供します。

チェック ThumbEE 拡張機能で提供されます。チェックは、特定のハンドラ エントリポイントへの無条件分岐を引き起こします。ThumbEE チェックハンドラのベースアドレスは、TEEHBR に保持されます。*ThumbEE ハンドラ ベースレジスタ (TEEHBR)* P. A2-71 を参照して下さい。

A2.11.1 Yield 命令

対称マルチスレッディング(SMT) 設計では、スレッドは Yield 命令を使用して、命令を実行するプロセッサにヒントを与えることができます。Yield ヒントは、スレッドで現在行われている動作の重要性が低く、譲歩できることを示します。例えば、スレッドがスピンロックのまま動いていない場合です。同様の動作は、マルチプロセッサ (MP) システムのスヌープバスのアービトレーション優先度を変更する際にも使用されることがあります。このような命令を定義することにより、SMT システムと SMP システムのバイナリ互換性が可能になります。

ARMv7 では、特定の NOP ヒント命令として、`YIELD` 命令が定義されています。*YIELD* P. A8-812 を参照して下さい。

`YIELD` 命令は、シングルスレッド システムでは効果がありませんが、そのようなシステムの開発者は、この命令を使用し、マルチプロセッサまたはマルチスレッディングシステムへの移行に際して、意図する使用法を示すことができます。オペレーティングシステムでは、実装上の利点がなければ `NOP` として扱われることを承知した上で、Yield ヒントが望ましい場所に `YIELD` を使用できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter A3
Application Level Memory Model

This chapter gives an application level view of the memory model. It contains the following sections:

- *Address space* on page A3-2
- *Alignment support* on page A3-4
- *Endian support* on page A3-7
- *Synchronization and semaphores* on page A3-12
- *Memory types and attributes and the memory order model* on page A3-24
- *Access rights* on page A3-38
- *Virtual and physical addressing* on page A3-40
- *Memory access order* on page A3-41
- *Caches and memory hierarchy* on page A3-51.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 A3 章
アプリケーションレベルのメモリモデル

本章では、アプリケーションレベルから見たメモリモデルについて説明します。本章は次のセクションから構成されています。

- *アドレス空間* P. A3-2
- *アライメントのサポート* P. A3-4
- *エンディアンのサポート* P. A3-7
- *同期化とセマフォ* P. A3-12
- *メモリのタイプおよび属性とメモリオーダ モデル* P. A3-24
- *アクセス権* P. A3-38
- *仮想アドレシングと物理アドレシング* P. A3-40
- *メモリアクセスの順序* P. A3-41
- *キャッシュとメモリ階層* P. A3-51

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.1 Address space

The ARM architecture uses a single, flat address space of 2^{32} 8-bit bytes. Byte addresses are treated as unsigned numbers, running from 0 to 2^{32} - 1. The address space is also regarded as:

- 2^{30} 32-bit words:
  - — the address of each word is word-aligned, meaning that the address is divisible by 4 and the last two bits of the address are 0b00
  - — the word at word-aligned address A consists of the four bytes with addresses A, A+1, A+2 and A+3.
- 2^{31} 16-bit halfwords:
  - — the address of each halfword is halfword-aligned, meaning that the address is divisible by 2 and the last bit of the address is 0
  - — the halfword at halfword-aligned address A consists of the two bytes with addresses A and A+1.

In some situations the ARM architecture supports accesses to halfwords and words that are not aligned to the appropriate access size, see *Alignment support* on page A3-4.

Normally, address calculations are performed using ordinary integer instructions. This means that the address wraps around if the calculation overflows or underflows the address space. Another way of describing this is that any address calculation is reduced modulo 2^{32}.

A3.1.1 Address incrementing and address space overflow

When a processor performs normal sequential execution of instructions, it effectively calculates:

```
(address_of_current_instruction) + (size_of_executed_instruction)
```

after each instruction to determine which instruction to execute next.

Note

The size of the executed instruction depends on the current instruction set, and might depend on the instruction executed.

If this address calculation overflows the top of the address space, the result is UNPREDICTABLE. In other words, a program must not rely on sequential execution of the instruction at address `0x00000000` after the instruction at address:

- `0xFFFFFFFC`, when a 4-byte instruction is executed
- `0xFFFFFFFE`, when a 2-byte instruction is executed
- `0xFFFFFFFF`, when a single byte instruction is executed.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A3.1 アドレス空間

ARM アーキテクチャでは、2^{32} 個の 8 ビットバイトで構成される、単一のフラットなアドレス空間が使用されます。バイトアドレスは、0 ～ 2^{32} − 1 の範囲の符号なし整数として扱われます。このアドレス空間は、次のように解釈することもできます。

- 2^{30} 個の 32 ビットワード
  - – 各ワードのアドレスはワードアラインドです。つまり、アドレスは 4 で割り切れ、アドレスの最後の 2 ビットは 0b00 です。
  - – ワードアラインドのアドレス A に存在するワードは、アドレス A、A + 1、A + 2、A + 3 にある 4 つのバイトで構成されます。
- 2^{31} 個の 16 ビットハーフワード
  - – 各ハーフワードのアドレスはハーフワードアラインドです。つまり、アドレスは 2 で割り切れ、アドレスの最後のビットは 0 です。
  - – ハーフワードアラインドのアドレス A に存在するハーフワードは、アドレス A、A + 1 にある 2 つのバイトで構成されます。

一部の状況では、ARM アーキテクチャは、適切なアクセスサイズにアラインしていないハーフワードやワードへのアクセスをサポートすることがあります。*アライメントのサポート* P. A3- 4 を参照して下さい。

アドレス計算は通常、普通の整数命令を使用して実行されます。これは、計算がアドレス空間をオーバフローまたはアンダーフローした場合にアドレスのラップアラウンドが行われることを意味します。言い換えると、すべてのアドレス計算はモジュロ 2^{32} に限定されます。

A3.1.1 アドレスのインクリメントとアドレス空間のオーバフロー

プロセッサが命令を通常にシーケンシャル実行する場合、実効上は次の計算が行われます。

```
(address_of_current_instruction) + (size_of_executed_instruction)
```

各命令の後でこの計算を行い、次に実行される命令が決定されます。

注

実行される命令のサイズは、現在の命令セットに依存し、実行される命令にも依存する場合があります。

このアドレス計算がアドレス空間の最上位をオーバフローする場合、結果は予測不能です。言い換えれば、プログラムは、次のアドレスで命令を実行した後で、アドレス `0x00000000` にある命令がシーケンシャルに実行されることを前提とできません。

- 4 バイト命令が実行される場合、`0xFFFFFFFC`
- 2 バイト命令が実行される場合、`0xFFFFFFFE`
- 1 バイト命令が実行される場合、`0xFFFFFFFF`

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

This UNPREDICTABLE behavior only applies to instructions that are executed, including those that fail their condition code check. Most ARM implementations prefetch instructions ahead of the currently-executing instruction. If this prefetching overflows the top of the address space, it does not cause UNPREDICTABLE behavior unless a prefetched instruction with an overflowed address is actually executed.

LDC, LDM, LDRD, POP, PUSH, STC, STRD, and STM instructions access a sequence of words at increasing memory addresses, effectively incrementing the memory address by 4 for each load or store. If this calculation overflows the top of the address space, the result is UNPREDICTABLE. In other words, programs must not use these instructions in such a way that they attempt to access the word at address 0x00000000 sequentially after the word at address 0xFFFFFFFC.

Note

In some cases instructions that operate on multiple words can decrement the memory address by 4 after each word access. If this calculation underflows the address space, by decrementing the address 0x00000000, the result is UNPREDICTABLE.

The behavior of any unaligned load or store with a calculated address that would access the byte at 0xFFFFFFFF and the byte at address 0x00000000 as part of the instruction is UNPREDICTABLE.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

この予測不能な動作は実行される命令にのみ適用されますが、これには条件コードチェックに失敗した命令も含まれます。ほとんどの ARM 実装では、現在実行中の命令よりも先の命令のプリフェッチを行います。このプリフェッチがアドレス空間の最上位をオーバフローする場合でも、アドレスがオーバフローしてプリフェッチされた命令が実際に実行されない限り、予測不能な動作は発生しません。

LDC、LDM、LDRD、POP、PUSH、STC、STRD、STM の各命令は、連続したメモリアドレスに存在する一連のワードにアクセスし、各レジスタのロード / ストアごとにメモリアドレスを 4 ずつインクリメントします。この計算がアドレス空間の最上位をオーバフローする場合、結果は予測不能です。つまり、プログラムでこれらの命令を使用する場合、アドレス 0xFFFFFFFC にあるワードの後で、アドレス 0x00000000 のアドレスにあるワードにシーケンシャルアクセスを試みるような使い方をしてはいけません。

注

場合によっては、複数のワードに作用する命令が、各ワードアクセスの後でメモリアドレスを 4 ずつデクリメントすることがあります。この計算で、アドレスが 0x00000000 からデクリメントされてアドレス空間をアンダーフローする場合、結果は予測不能です。

アンアラインドのロードまたはストアで、命令によって計算されたアドレスにより、0xFFFFFFFF のバイトと 0x00000000 のバイトへのアクセスが行われる場合、動作は予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.2 Alignment support

Instructions in the ARM architecture are aligned as follows:

- ARM instructions are word-aligned
- Thumb and ThumbEE instructions are halfword-aligned
- Java bytecodes are byte-aligned.

The data alignment behavior supported by the ARM architecture has changed significantly between ARMv4 and ARMv7. This behavior is indicated by the SCTLR.U bit, see:

- *c1, System Control Register (SCTLR)* on page B3-96 for a VMSAv7 implementation
- *c1, System Control Register (SCTLR)* on page B4-45 for a PMSAv7 implementation
- *c1, System Control Register (SCTLR)* on page AppxG-34 for architecture versions before ARMv7.

This bit defines the alignment behavior of the memory system for data accesses. Table A3-1 shows the values of SCTLR.U for the different architecture versions.

Table A3-1 SCTLR.U bit values for different architecture versions

Architecture version	SCTLR.U value
Before ARMv6	0
ARMv6	0 or 1
ARMv7	1

On an ARMv6 processor, the SCTLR.U bit indicates which of two possible alignment models is selected:

U == 0 The processor implements the legacy alignment model. This is described in *Alignment* on page AppxG-6.

Note

The use of U == 0 is deprecated in ARMv6T2, and is obsolete from ARMv7.

U == 1 The processor implements the alignment model described in this section. This model supports unaligned data accesses.

ARMv7 requires the processor to implement the alignment model described in this section.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A3.2 アライメントのサポート

ARM アーキテクチャにおける命令は次のようにアラインしています。

- ARM 命令はワードアラインです。
- Thumb および ThumbEE 命令はハーフワードアラインです。
- Java バイトコードはバイトアラインです。

ARM アーキテクチャがサポートするデータアライメント動作は、ARMv4 と ARMv7 の間で大きく変更されています。この動作は SCTLR.U ビットによって示されます。次のセクションを参照して下さい。

- *c1、システム制御レジスタ (SCTLR)* P. B3-96（VMSAv7 実装の場合）
- *c1、システム制御レジスタ (SCTLR)* P. B4-45（PMSAv7 実装の場合）
- *c1、システム制御レジスタ (SCTLR)* P. 付録 G-34 （ARMv7 以前のアーキテクチャバージョンの場合）

このビットは、データアクセスについてメモリシステムのアライメント動作を定義します。各アーキテクチャバージョンにおける SCTLR.U の値を、表 A3-1 に示します。

表 A3-1 各アーキテクチャバージョンにおける SCTLR.U ビットの値

アーキテクチャのバージョン	SCTLR.U の値
ARMv6 以前	0
ARMv6	0 または 1
ARMv7	1

ARMv6 プロセッサでは、SCTLR.U ビットは 2 つのアライメントモデルのうちどちらが選択されているかを示します。

U == 0 プロセッサには、レガシーアライメント モデルが実装されています。これについては、*アライメント* P. 付録 G-6 を参照して下さい。

注

ARMv6T2 における U == 0 の使用は推奨されません。ARMv7 以降は廃止されています。

U == 1 プロセッサには、このセクションで説明するアライメントモデルが実装されています。このモデルはアンアラインド データアクセスをサポートしています。

ARMv7 では、プロセッサがこのセクションで説明するアライメントモデルを実装している必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.2.1 Unaligned data access

An ARMv7 implementation must support unaligned data accesses. The SCTLR.U bit is RAO to indicate this support. The SCTLR.A bit, the strict alignment bit, controls whether strict alignment is required. The checking of load and store alignment depends on the value of this bit. For more information, see *c1, System Control Register (SCTLR)* on page B3-96 for a VMSA implementation, or *c1, System Control Register (SCTLR)* on page B4-45 for a PMSA implementation.

Table A3-2 shows how the checking of load and store alignment depends on the instruction type and the value of SCTLR.A.

Table A3-2 Alignment requirements of load/store instructions

Instructions	Alignment check	Result if check fails when:	
		SCTLR.A == 0	SCTLR.A == 1
LDRB, LDREXB, LDRBT, LDRSB, LDRSBT, STRB, STREXB, STRBT, SWPB, TBB	None	-	-
LDRH, LDRHT, LDRSH, LDRSHT, STRH, STRHT, TBH	Halfword	Unaligned access	Alignment fault
LDREXH, STREXH	Halfword	Alignment fault	Alignment fault
LDR, LDRT, STR, STRT	Word	Unaligned access	Alignment fault
LDREX, STREX	Word	Alignment fault	Alignment fault
LDREXD, STREXD	Doubleword	Alignment fault	Alignment fault
All forms of LDM, LDRD, PUSH, POP, RFE, SRS, all forms of STM, STRD, SWP	Word	Alignment fault	Alignment fault
LDC, LDC2, STC, STC2	Word	Alignment fault	Alignment fault
VLDM, VLDR, VSTM, VSTR	Word	Alignment fault	Alignment fault
VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, all with standard alignment[a]	Element size	Unaligned access	Alignment fault
VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, all with @<align> specified[a]	As specified by @<align>	Alignment fault	Alignment fault

a. These element and structure load&store instructions are only in the Advanced SIMD extension to the ARMv7 ARM and Thumb instruction sets. ARMv7 does not support the pre-ARMv6 alignment model, so you cannot use that model with these instructions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.2.1 アンアラインド データアクセス

ARMv7 の実装は、アンアラインド データアクセスをサポートしている必要があります。このサポートを示すため、SCTLR.U ビットは RAO です。厳密アライメントビットである SCTLR.A ビットは、厳密なアライメントが必要かどうかを制御します。ロードおよびストアアライメントのチェックは、このビットの値によって決まります。詳細については、*c1、システム制御レジスタ (SCTLR)* P. B3-96（VMSA 実装の場合）、または *c1、システム制御レジスタ (SCTLR)* P. B4-45（PMSA 実装の場合）を参照して下さい。

ロードおよびストアアライメントのチェックが、命令のタイプと SCTLR.A ビットの値によってどのように決定されるかを、表 A3-2 に示します。

表 A3-2 ロード / ストア命令のアライメント条件

命令	アライメントチェック	チェックが失敗した場合の結果	
		SCTLR.A = 0	SCTLR.A = 1
LDRB、LDREXB、LDRBT、LDRSB、LDRSBT、STRB、STREXB、STRBT、SWPB、TBB	なし	–	–
LDRH、LDRHT、LDRSH、LDRSHT、STRH、STRHT、TBH	ハーフワード	アンアラインドアクセス	アライメントフォールト
LDREXH、STREXH	ハーフワード	アライメントフォールト	アライメントフォールト
LDR、LDRT、STR、STRT	ワード	アンアラインドアクセス	アライメントフォールト
LDREX、STREX	ワード	アライメントフォールト	アライメントフォールト
LDREXD、STREXD	ダブルワード	アライメントフォールト	アライメントフォールト
LDM、LDRD、PUSH、POP、RFE、SRS のすべての形式、STM、STRD、SWP のすべての形式	ワード	アライメントフォールト	アライメントフォールト
LDC、LDC2、STC、STC2	ワード	アライメントフォールト	アライメントフォールト
VLDM、VLDR、VSTM、VSTR	ワード	アライメントフォールト	アライメントフォールト
VLD1、VLD2、VLD3、VLD4、VST1、VST2、VST3、VST4（すべて標準アライメント）[a]	エレメントサイズ	アンアラインドアクセス	アライメントフォールト
VLD1、VLD2、VLD3、VLD4、VST1、VST2、VST3、VST4（すべて @<align> 指定）[a]	@<align> の指定どおり	アライメントフォールト	アライメントフォールト

a. これらのエレメントと構造のロード / ストア命令は、ARMv7 の ARM および Thumb 命令セットのアドバンスト SIMD 拡張機能にのみ存在します。ARMv7 は ARMv6 より前のアライメントモデルをサポートしないため、それらのモデルにはここに示した命令を使用できません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.2.2 Cases where unaligned accesses are UNPREDICTABLE

The following cases cause the resulting unaligned accesses to be UNPREDICTABLE, and overrule any successful load or store behavior described in *Unaligned data access* on page A3-5:

- Any load instruction that is not faulted by the alignment restrictions and that loads the PC has UNPREDICTABLE behavior if it the address it loads from is not word-aligned.
- Any unaligned access that is not faulted by the alignment restrictions and that accesses memory with the Strongly-ordered or Device attribute has UNPREDICTABLE behavior.

Note

These memory attributes are described in *Memory types and attributes and the memory order model* on page A3-24.

A3.2.3 Unaligned data access restrictions in ARMv7 and ARMv6

ARMv7 and ARMv6 have the following restrictions on unaligned data accesses:

- Accesses are not guaranteed to be single-copy atomic, see *Atomicity in the ARM architecture* on page A3-26. An access can be synthesized out of a series of aligned operations in a shared memory system without guaranteeing locked transaction cycles.
- Unaligned accesses typically take a number of additional cycles to complete compared to a naturally aligned transfer. The real-time implications must be analyzed carefully and key data structures might need to have their alignment adjusted for optimum performance.
- If an unaligned access occurs across a page boundary, the operation can abort on either or both halves of the access.

Shared memory schemes must not rely on seeing monotonic updates of non-aligned data of loads and stores for data items larger than byte wide. For more information, see *Atomicity in the ARM architecture* on page A3-26.

Unaligned access operations must not be used for accessing Device memory-mapped registers. They must only be used with care in shared memory structures that are protected by aligned semaphores or synchronization variables.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A3.2.2 アンアラインドアクセスが予測不能となるケース

次のケースでは、アンアラインドアクセスの結果が予測不能となり、*アンアラインド データアクセス* P. A3- 5 に説明されている正常なロードまたはストア動作は行われません。

- アライメント制限によってフォールトとされていないロード命令で、PC をロードする場合、ロード元のアドレスがワードアラインしていない場合の動作は予測不能です。
- アライメント制限によってフォールトとされていないアンアラインドアクセスで、ストロングリオーダまたはデバイス属性のメモリに対してアクセスが行われた場合、動作は予測不能です。

注

これらのメモリ属性については、*メモリのタイプおよび属性とメモリオーダ モデル* P. A3- 24 に説明されています。

A3.2.3 ARMv7 および ARMv6 におけるアンアラインド データアクセスの制限

ARMv7 および ARMv6 でのアンアラインド データアクセスには、次の制限が適用されます。

- シングルコピー アトミックなアクセスは保証されません。*ARM アーキテクチャにおけるアトミック性* P. A3- 26 を参照して下さい。トランザクションサイクルのロックの保証なしに、共有メモリシステム内のアラインな一連の操作によってアクセスが合成される可能性があります。
- 一般に、自然アラインしている転送と比較して、アンアラインドアクセスの完了には多くのサイクルが必要です。リアルタイム性が必要な状況を十分に分析し、重要なデータ構造についてはパフォーマンスが最適になるようにアライメントを調整することが必要な可能性があります。
- ページ境界にまたがるアンアラインドアクセスが生じた場合、そのアクセスの片方または両方でアボートが発生する可能性があります。

共有メモリスキーマでは、バイト幅より大きいデータ項目について、アンアラインドデータのロードとストアの単調的な更新を前提とできません。詳細については、*ARM アーキテクチャにおけるアトミック性* P. A3- 26 を参照して下さい。

アンアラインドアクセス動作は、デバイスメモリマップ レジスタへのアクセスには使用できません。アラインド セマフォや同期変数によって保護された共有メモリ構造においてのみ、慎重に使用する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.3 Endian support

The rules in *Address space* on page A3-2 require that for a word-aligned address A:

- the word at address A consists of the bytes at addresses A, A+1, A+2 and A+3
- the halfword at address A consists of the bytes at addresses A and A+1
- the halfword at address A+2 consists of the bytes at addresses A+2 and A+3.
- the word at address A therefore consists of the halfwords at addresses A and A+2.

However, this does not specify completely the mappings between words, halfwords, and bytes.

A memory system uses one of the two following mapping schemes. This choice is known as the endianness of the memory system.

In a *little-endian* memory system:

- the byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address
- the byte at a halfword-aligned address is the least significant byte in the halfword at that address.

In a *big-endian* memory system:

- the byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address
- the byte at a halfword-aligned address is the most significant byte in the halfword at that address.

For a word-aligned address A, Table A3-3 and Table A3-4 on page A3-8 show the relationship between:

- the word at address A
- the halfwords at addresses A and A+2
- the bytes at addresses A, A+1, A+2 and A+3.

Table A3-3 shows this relationship for a big-endian memory system, and Table A3-4 on page A3-8 shows the relationship for a little-endian memory system.

Table A3-3 Big-endian memory system

MSByte	MSByte - 1	LSByte + 1	LSByte
Word at Address A			
Halfword at Address A		Halfword at Address A+2	
Byte at Address A	Byte at Address A+1	Byte at Address A+2	Byte at Address A+3

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.3 エンディアンのサポート

アドレス空間 P. A3-2 に記載されている規則では、ワードアラインドのアドレス A が次のように定義されます。

- アドレス A のワードは、アドレス A、A + 1、A + 2、A + 3 のバイトから構成されます。
- アドレス A のハーフワードは、アドレス A および A + 1 のバイトから構成されます。
- アドレス A + 2 のハーフワードは、アドレス A + 2 および A + 3 のバイトから構成されます。
- 結果として、アドレス A のワードは、アドレス A および A + 2 のハーフワードから構成されます。

ただし、この定義ではワード、ハーフワード、バイト間のマッピングが完全には指定されていません。

メモリシステムでは、次の 2 つのマッピング方式のいずれかが使用されます。この選択は、メモリシステムのエンディアン形式と呼ばれます。

*リトルエンディアン*のメモリシステムの場合

- ワードアラインされたアドレスのバイトまたはハーフワードが、そのアドレスにあるワード内の最下位バイトまたはハーフワードです。
- ハーフワードアラインされたアドレスのバイトが、そのアドレスにあるハーフワード内の最下位バイトです。

*ビッグエンディアン*のメモリシステムの場合

- ワードアラインされたアドレスのバイトまたはハーフワードが、そのアドレスのワード内で最上位のバイトまたはハーフワードです。
- ハーフワードアラインされたアドレスのバイトが、そのアドレスにあるハーフワード内の最上位バイトです。

ワードアラインアドレス A について、表 A3-3 と P. A3-8 の表 A3-4 は次のエレメントの相互関係を示しています。

- アドレス A のワード
- アドレス A および A + 2 のハーフワード
- アドレス A、A + 1、A + 2、A + 3 のバイト

表 A3-3 はビッグエンディアンのメモリシステム、P. A3-8 の表 A3-4 はリトルエンディアンのメモリシステムでの関係です。

表 A3-3 ビッグエンディアンのメモリシステム

MSByte	MSByte − 1	LSByte + 1	LSByte
アドレス A のワード			
アドレス A のハーフワード		アドレス A + 2 のハーフワード	
アドレス A のバイト	アドレス A+1 のバイト	アドレス A+2 のバイト	アドレス A+3 のバイト

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A3-4 Little-endian memory system

MSByte	MSByte - 1	LSByte + 1	LSByte
Word at Address A			
Halfword at Address A+2		Halfword at Address A	
Byte at Address A+3	Byte at Address A+2	Byte at Address A+1	Byte at Address A

The big-endian and little-endian mapping schemes determine the order in which the bytes of a word or halfword are interpreted. For example, a load of a word (4 bytes) from address `0x1000` always results in an access of the bytes at memory locations `0x1000`, `0x1001`, `0x1002`, and `0x1003`. The endianness mapping scheme determines the significance of these four bytes.

A3.3.1 Control of the endianness mapping scheme in ARMv7

In ARMv7-A, the mapping of instruction memory is always little-endian. In ARMv7-R, instruction endianness can be controlled at the system level, see *Instruction endianness*.

For information about data memory endianness control, see *ENDIANSTATE* on page A2-19.

Note

Versions of the ARM architecture before ARMv7 had a different mechanism to control the endianness, see *Endian configuration and control* on page AppxG-20.

A3.3.2 Instruction endianness

Before ARMv7, the ARM architecture included legacy support for an alternative big-endian memory model, described as BE-32 and controlled by the B bit, bit [7], of the SCTLR, see *c1, System Control Register (SCTLR)* on page AppxG-34. ARMv7 does not support BE-32 operation, and bit [7] of the SCTLR is RAZ.

Where legacy object code for ARM processors contains instructions with a big-endian byte order, the removal of support for BE-32 operation requires the instructions in the object files to have their bytes reversed for the code to be executed on an ARMv7 processor. This means that:

- each Thumb instruction, whether a 32-bit Thumb instruction or a 16-bit Thumb instruction, must have the byte order of each halfword of instruction reversed
- each ARM instruction must have the byte order of each word of instruction reversed.

For most situations, this can be handled in the link stage of a tool-flow, provided the object files include sufficient information to permit this to happen. In practice, this is the situation for all applications with the ARMv7-A profile.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A3-4 リトルエンディアンのメモリシステム

MSByte	MSByte − 1	LSByte + 1	LSByte
アドレス A のワード			
アドレス A + 2 のハーフワード		アドレス A のハーフワード	
アドレス A+3 のバイト	アドレス A+2 のバイト	アドレス A+1 のバイト	アドレス A のバイト

ビッグエンディアンおよびリトルエンディアンのマッピング方式により、ワードまたはハーフワードのバイトが解釈される順序が決定されます。例えば、アドレス `0x1000` からのワード（4 バイト）のロードでは常に、メモリ位置 `0x1000`、`0x1001`、`0x1002`、`0x1003` に含まれているバイトがアクセスされます。エンディアンのマッピング方式により、この 4 バイトの上位と下位の関係が決定されます。

A3.3.1 ARMv7 におけるエンディアンマッピング方式の制御

ARMv7-A では、命令メモリのマッピングは常にリトルエンディアンです。ARMv7-R では、命令のエンディアン形式をシステムレベルで制御できます。*命令のエンディアン形式*を参照して下さい。

データメモリのエンディアン制御については、*ENDIANSTATE* P. A2-19 を参照して下さい。

注

ARMv7 より前の ARM アーキテクチャバージョンでは、エンディアン形式の制御機構が異なっています。*エンディアン構成と制御* P. 付録 G-20 を参照して下さい。

A3.3.2 命令のエンディアン形式

ARMv7 以前の ARM アーキテクチャには、代替ビッグエンディアン メモリモデルの過去のサポートが含まれます。これは、BE-32 と呼ばれ、SCTLR の B ビット（ビット [7]）で制御されます。*c1、システム制御レジスタ (SCTLR)* P. 付録 G-34 を参照して下さい。ARMv7 は BE-32 動作をサポートしません。SCTLR のビット [7] は RAZ です。

ARM プロセッサのレガシー オブジェクトコードには、ビッグエンディアンのバイト順序の命令が含まれます。BE-32 動作のサポートを削除するには、ARMv7 プロセッサで実行されるコードについて、オブジェクトファイルに入っている命令のバイトを反転させる必要があります。これは次のことを意味します。

- 32 ビット Thumb 命令であれ、16 ビット Thumb 命令であれ、すべての Thumb 命令について各ハーフワードのバイト順序を反転させる必要があります。
- すべての ARM 命令について、命令の各ワードのバイト順序を反転させる必要があります。

これはほとんどの場合、オブジェクトファイルにこれを実行するための十分な情報が含まれている限り、ツールフローのリンクステージで処理できます。実際には、これが ARMv7-A プロファイルを持つすべてのアプリケーションについて該当します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For applications of the ARMv7-R profile, there are some legacy code situations where the arrangement of the bytes in the object files cannot be adjusted by the linker. For these object files to be used by an ARMv7-R processor the byte order of the instructions must be reversed by the processor at runtime. Therefore, the ARMv7-R profile permits configuration of the instruction endianness.

Instruction endianness static configuration, ARMv7-R only

To provide support for legacy big-endian object code, the ARMv7-R profile supports optional byte order reversal hardware as a static option from reset. The ARMv7-R profile includes a read-only bit in the CP15 Control Register, SCTLR.IE, bit [31]. For more information, see *c1, System Control Register (SCTLR)* on page B4-45.

A3.3.3 Element size and endianness

The effect of the endianness mapping on data transfers depends on the size of the data element or elements transferred by the load/store instructions. Table A3-5 lists the element sizes of all the load/store instructions, for all instruction sets.

Table A3-5 Element size of load/store instructions

Instructions	Element size
LDRB, LDREXB, LDRBT, LDRSB, LDRSBT, STRB, STREXB, STRBT, SWPB, TBB	Byte
LDRH, LDREXH, LDRHT, LDRSH, LDRSHT, STRH, STREXH, STRHT, TBH	Halfword
LDR, LDRT, LDREX, STR, STRT, STREX	Word
LDRD, LDREXD, STRD, STREXD	Word
All forms of LDM, PUSH, POP, RFE, SRS, all forms of STM, SWP	Word
LDC, LDC2, STC, STC2, VLDM, VLDR, VSTM, VSTR	Word
VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4	Element size of the Advanced SIMD access

A3.3.4 Instructions to reverse bytes in a general-purpose register

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the processor requires an efficient method to transform explicitly the endianness of the data.

In ARMv7, the ARM and Thumb instruction sets provide this functionality. There are instructions to:

- Reverse word (four bytes) register, for transforming big and little-endian 32-bit representations. See *REV* on page A8-272.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARMv7-R プロファイルのアプリケーションについては、一部の過去のコードにおいて、オブジェクトファイル内のバイトの配置をリンカで調整できないことがあります。このようなオブジェクトファイルを ARMv7-R プロセッサで使用するには、実行時にプロセッサで命令のバイト順序を反転する必要があります。このため、ARMv7-R のプロファイルでは、命令のエンディアン形式の構成が許可されています。

命令のエンディアン形式の静的な構成（ARMv7-R のみ）

従来のビッグエンディアン オブジェクトコードをサポートするため、ARMv7-R プロファイルは、リセット時からの静的オプションとして、バイト順序反転ハードウェアをサポートしています。ARMv7-R プロファイルには、CP15 制御レジスタに読み出し専用ビット SCTLR.IE（ビット [31]）が含まれます。詳細については、*c1、システム制御レジスタ (SCTLR)* P. B4-45 を参照して下さい。

A3.3.3 エレメントサイズとエンディアン形式

エンディアン形式マッピングがデータ転送にどのように影響するかは、ロード / ストア命令で転送されるデータエレメントのサイズによって変わります。すべての命令セットの各ロード / ストア命令のエレメントサイズを、表 A3-5 に示します。

表 A3-5 ロード / ストア命令のエレメントサイズ

命令	エレメントサイズ
LDRB、LDREXB、LDRBT、LDRSB、LDRSBT、STRB、STREXB、STRBT、SWPB、TBB	バイト
LDRH、LDREXH、LDRHT、LDRSH、LDRSHT、STRH、STREXH、STRHT、TBH	ハーフワード
LDR、LDRT、LDREX、STR、STRT、STREX	ワード
LDRD、LDREXD、STRD、STREXD	ワード
LDM、PUSH、POP、RFE、SRS のすべての形式、STM、SWP のすべての形式	ワード
LDC、LDC2、STC、STC2、VLDM、VLDR、VSTM、VSTR	ワード
VLD1、VLD2、VLD3、VLD4、VST1、VST2、VST3、VST4	アドバンスト SIMD アクセスのエレメントサイズ

A3.3.4 汎用レジスタ内でバイトを反転する命令

アプリケーションまたはデバイスドライバは、内部データ構造とエンディアン形式の異なるメモリマップ ペリフェラルレジスタや、共有メモリ構造に接続することが必要があります。同様に、オペレーティングシステムのエンディアン形式も、ペリフェラルレジスタや共有メモリと一致していない場合があります。このような場合、データのエンディアン形式を明示的に変換する効率的な方法を、プロセッサに搭載する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- Reverse halfword and sign-extend, for transforming signed 16-bit representations. See *REVSH* on page A8-276.
- Reverse packed halfwords in a register for transforming big- and little-endian 16-bit representations. See *REV16* on page A8-274.

A3.3.5 Endianness in Advanced SIMD

Advanced SIMD element load/store instructions transfer vectors of elements between memory and the Advanced SIMD register bank. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used by the processor to load and store data correctly in both big-endian and little-endian systems.

Consider. for example, the instruction:

```
VLD1.16 {D0}, [R1]
```

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in Figure A3-1. Therefore, the order of the elements in the registers is the same regardless of the endianness configuration. This means that Advanced SIMD code is usually independent of endianness.

Figure A3-1 Advanced SIMD byte order example

The Advanced SIMD extension supports Little-Endian (LE) and Big-Endian (BE) models.

For information about the alignment of Advanced SIMD instructions see *Unaligned data access* on page A3-5.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ARMv7 では、ARM 命令セットと Thumb 命令セットがこの機能を持っています。この機能には、次のような命令が含まれています。

- ワード（4 バイト）レジスタ反転命令。ビッグ / リトルエンディアンの 32 ビット表現の変換に使用します。詳細については、*REV* P. A8-272 を参照して下さい。
- ハーフワードの符号付き 16 ビット表現を反転し、符号拡張を行う命令。詳細については、*REVSH* P. A8-276 を参照して下さい。
- レジスタ内のパック化ハーフワード反転命令。ビッグ / リトルエンディアンの 16 ビット表現の変換に使用します。詳細については、*REV16* P. A8-274 を参照して下さい。

A3.3.5 アドバンスト SIMD でのエンディアン形式

アドバンスト SIMD のエレメントロード / ストア命令は、メモリとアドバンスト SIMD レジスタバンクとの間でエレメントのベクタを転送します。命令は、転送の長さと転送するデータエレメントのサイズの両方を指定します。この情報は、プロセッサがビッグエンディアン システムとリトルエンディアン システムの両方でデータを正しくロード / ストアするために使用されします。

例えば、次のような命令について考えます。

```
VLD1.16 {D0}, [R1]
```

この命令は、64 ビットレジスタに 4 つの 16 ビット値をロードします。4 つのエレメントは配列順にレジスタに格納されます。インデクスの最も低いエレメントは最下位のアドレスからフェッチされます。エレメントのバイト順序は、図 A3-1 のように、エンディアン形式の構成によって決まります。したがって、レジスタ内のエレメントの順序は、エンディアン形式の構成に関係なく同じです。つまり、アドバンスト SIMD のコードは、通常はエンディアン形式の影響を受けません。

図 A3-1 アドバンスト SIMD のバイト順序の例

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

Advanced SIMD is an extension to the ARMv7 ARM and Thumb instruction sets. In ARMv7, the SCTLR.B bit always has the value 0, indicating that ARMv7 does not support the legacy BE-32 endianness model, and you cannot use this model with Advanced SIMD element and structure load/store instructions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アドバンスト SIMD 拡張機能は、リトルエンディアン (LE) とビッグエンディアン (BE) のモデルをサポートします。

アドバンスト SIMD 命令のアライメントについては、*アンアラインド データアクセス* P. A3- 5 を参照して下さい。

注

アドバンスト SIMD は、ARMv7 の ARM および Thumb 命令セットの拡張機能です。ARMv7 では、SCTLR.B ビットの値は常に 0 で、ARMv7 が過去の BE-32 エンディアン形式モデルをサポートしないことを示しています。アドバンスト SIMD のエレメントや構造のロード / ストア命令には、このモデルを使用できません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.4 Synchronization and semaphores

In architecture versions before ARMv6, support for the synchronization of shared memory depends on the SWP and SWPB instructions. These are read-locked-write operations that swap register contents with memory, and are described in *SWP, SWPB* on page A8-432. These instructions support basic busy/free semaphore mechanisms, but do not support mechanisms that require calculation to be performed on the semaphore between the read and write phases.

ARMv6 introduced a new mechanism to support more comprehensive non-blocking synchronization of shared memory, using *synchronization primitives* that scale for multiprocessor system designs. ARMv6 provided a pair of synchronization primitives, LDREX and STREX. ARMv7 extends the new model by:

- adding byte, halfword and doubleword versions of the synchronization primitives
- adding a Clear-Exclusive instruction, CLREX
- adding the synchronization primitives to the Thumb instruction set.

Note

From ARMv6, use of the SWP and SWPB instructions is deprecated. ARM strongly recommends that all software migrates to using the new synchronization primitives described in this section.

In ARMv7, the synchronization primitives provided in the ARM and Thumb instruction sets are:

- Load-Exclusives:
  - LDREX, see *LDREX* on page A8-142
  - LDREXB, see *LDREXB* on page A8-144
  - LDREXD, see *LDREXD* on page A8-146
  - LDREXH, see *LDREXH* on page A8-148
- Store-Exclusives:
  - STREX, see *STREX* on page A8-400
  - STREXB, see *STREXB* on page A8-402
  - STREXD, see *STREXD* on page A8-404
  - STREXH, see *STREXH* on page A8-406
- Clear-Exclusive, CLREX, see *CLREX* on page A8-70.

Note

This section describes the operation of a Load-Exclusive/Store-Exclusive pair of synchronization primitives using, as examples, the LDREX and STREX instructions. The same description applies to any other pair of synchronization primitives:

- LDREXB used with STREXB
- LDREXD used with STREXD
- LDREXH used with STREXH.

Each Load-Exclusive instruction must be used only with the corresponding Store-Exclusive instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A3.4 同期化とセマフォ

ARMv6 以前のアーキテクチャバージョンでは、共有メモリの同期化のサポートが SWP 命令と SWPB 命令に依存します。これらの命令は読み出し－ロック－書き込み命令で、レジスタの内容をメモリとスワップします。これについては *SWP*、*SWPB* P. A8- 432 に説明されています。これらの命令は、基本的なビジー / フリーのセマフォ機構をサポートしていますが、読み出しフェーズと書き込みフェーズの間でセマフォに対して計算を実行する必要のある機構はサポートしません。

ARMv6 では、共有メモリのさらに包括的な非ブロッキング同期化をサポートする新しい機構が導入されました。これは、マルチプロセッサシステム設計用にスケーリング可能な*同期化基本命令*を使用します。ARMv6 には、LDREX および STREX の同期化基本命令がありました。ARMv7 では、この新しいモデルが次のように拡張されています。

- 同期化基本命令のバイト、ハーフワード、ダブルワードバージョンが追加されています。
- 排他クリア命令 CLREX が追加されています。
- Thumb 命令セットに、同期化基本命令が追加されています。

注

ARMv6 およびそれ以降では、SWP および SWPB 命令の使用が推奨されていません。すべてのソフトウェアについて、このセクションで説明する新しい同期化基本命令の使用に移行することを強く推奨します。

ARMv7 で、ARM および Thumb 命令セットに含まれている同期化基本命令は、次のとおりです。

- 排他ロード
  - LDREX。*LDREX* P. A8- 142 を参照して下さい。
  - LDREXB。*LDREXB* P. A8- 144 を参照して下さい。
  - LDREXD。*LDREXD* P. A8- 146 を参照して下さい。
  - LDREXH。*LDREXH* P. A8- 148 を参照して下さい。
- 排他ストア
  - STREX。*STREX* P. A8- 400 を参照して下さい。
  - STREXB。*STREXB* P. A8- 402 を参照して下さい。
  - STREXD。*STREXD* P. A8- 404 を参照して下さい。
  - STREXH。*STREXH* P. A8- 406 を参照して下さい。
- 排他クリア、CLREX。*CLREX* P. A8- 70 を参照して下さい。

注

このセクションでは、LDREX および STREX 命令など、同期化基本命令のペアである排他ロード / 排他ストア操作について説明します。同じ説明は、他の同期化基本命令のペアについても適用されます。

- LDREXB は、STREXB と対で使用されます。
- LDREXD は、STREXD と対で使用されます。
- LDREXH は、STREXH と対で使用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The model for the use of a Load-Exclusive/Store-Exclusive instruction pair, accessing a non-aborting memory address x is:

- The Load-Exclusive instruction reads a value from memory address x.
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address x only if no other observer, process, or thread has performed a more recent store of address x. The Store-Exclusive operation returns a status bit that indicates whether the memory write succeeded.

A Load-Exclusive instruction tags a small block of memory for exclusive access. The size of the tagged block is IMPLEMENTATION DEFINED, see *Tagging and the size of the tagged memory block* on page A3-20. A Store-Exclusive instruction to the same address clears the tag.

Note

In this section, the term processor includes any observer that can generate a Load-Exclusive or a Store-Exclusive.

A3.4.1 Exclusive access instructions and Non-shareable memory regions

For memory regions that do not have the *Shareable* attribute, the exclusive access instructions rely on a *local monitor* that tags any address from which the processor executes a Load-Exclusive. Any non-aborted attempt by the same processor to use a Store-Exclusive to modify any address is guaranteed to clear the tag.

A Load-Exclusive performs a load from memory, and:

- the executing processor tags the physical memory address for exclusive access
- the local monitor of the executing processor transitions to its Exclusive Access state.

A Store-Exclusive performs a conditional store to memory, that depends on the state of the local monitor:

If the local monitor is in its Exclusive Access state

- If the address of the Store-Exclusive is the same as the address that has been tagged in the monitor by an earlier Load-Exclusive, then the store takes place, otherwise it is IMPLEMENTATION DEFINED whether the store takes place.
- A status value is returned to a register:
  - — if the store took place the status value is 0
  - — otherwise, the status value is 1.
- The local monitor of the executing processor transitions to its Open Access state.

If the local monitor is in its Open Access state

- no store takes place
- a status value of 1 is returned to a register.
- the local monitor remains in its Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

各排他ロード命令は、対応する排他ストア命令とペアでのみ使用できます。

排他ロード / 排他ストア命令ペアの使用法のモデルでは、メモリアドレス x へのアボートしない読み出し / 書き込みは次のように行われます。

- 排他ロード命令は、メモリアドレス x から値を読み出します。
- 対応する排他ストア命令は、排他ロードによる読み出しの後に、他のオブザーバ、プロセス、またはスレッドがアドレス x へストアを実行していない場合に限り、メモリアドレス x への書き込みに成功します。排他ストア操作では、メモリ書き込みに成功したかどうかを示すステータスビットが返されます。

排他ロード命令では、メモリの小さいブロックに対して、排他アクセスを示すタグが付けられます。タグ付けされるブロックのサイズは実装定義です。詳細については、*タグ付けとタグ付けされたメモリブロックのサイズ* P. A3-20 を参照して下さい。このタグは、同じアドレスへの排他ストア命令によりクリアされます。

注

このセクションにおいて、プロセッサという言葉は、排他ロードまたは排他ストアを生成できるオブザーバすべてを含みます。

A3.4.1 排他アクセス命令と共有不可メモリ領域

*共有可能*属性を持たないメモリ領域の場合、排他アクセス命令は、プロセッサが排他ロードを実行するアドレスにタグを付ける *ローカルモニタ*に依存します。同じプロセッサがいずれかのアドレスを変更するために排他ストアを使用した場合、アボートが発生しない限り、そのタグは必ずクリアされます。

排他ロードは、メモリからロードを行うとともに次の処理を実行します。

- 実行中のプロセッサにより、その物理メモリアドレスに対して、排他アクセスのタグ付けが行われます。
- 実行中のプロセッサのローカルモニタが、排他アクセス状態へ遷移します。

排他ストア命令は、メモリに対して条件付きストアを実行します。このとき、ローカルモニタの状態に応じて次の動作が行われます。

ローカルモニタが排他アクセス状態の場合

- 排他ストアのアドレスが、以前の排他ロードによってモニタ内にタグ付けされたアドレスと同じである場合は、ストアが実行されます。それ以外の場合、ストアが実行されるかどうかは実装定義です。
- 次のように、ステータス値がレジスタに返されます。
  - – ストアが実行された場合、0
  - – それ以外の場合、1
- 実行中のプロセッサのローカルモニタが、オープンアクセス状態へ遷移します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When a processor writes using any instruction other than a Store-Exclusive:

- if the write is to a physical address that is not covered by its local monitor the write does not affect the state of the local monitor
- if the write is to a physical address that is covered by its local monitor it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

If the local monitor is in its Exclusive Access state and the processor performs a Store-Exclusive to any address other than the last one from which it performed a Load-Exclusive, it is IMPLEMENTATION DEFINED whether the store updates memory, but in all cases the local monitor is reset to its Open Access state. This mechanism:

- is used on a context switch, see *Context switch support* on page A3-21
- must be treated as a software programming error in all other cases.

Note

It is UNPREDICTABLE whether a store to a tagged physical address causes a tag in the local monitor to be cleared if that store is by an observer other than the one that caused the physical address to be tagged.

Figure A3-2 shows the state machine for the local monitor. Table A3-6 on page A3-15 shows the effect of each of the operations shown in the figure.

Operations marked * are possible alternative IMPLEMENTATION DEFINED options.

In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the tagged address to the most significant bits of the address x used for the operation. For more information see the section *Size of the tagged memory block*.

Figure A3-2 Local monitor state machine diagram

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ローカルモニタがオープンアクセス状態の場合

- ストアは実行されません。
- ステータス値 1 がレジスタに返されます。
- ローカルモニタはオープンアクセス状態のままです。

ステータス値が返されるレジスタは、排他ストア命令によって定義されます。

プロセッサが排他ストア以外の命令を使用して書き込みを行う場合、次の動作が行われます。

- ローカルモニタが処理を担当しない物理アドレスに書き込みが行われる場合、その書き込みによってローカルモニタの状態は影響を受けません。
- ローカルモニタが処理を担当する物理アドレスに書き込みが行われる場合、その書き込みによってローカルモニタの状態が影響を受けるかどうかは実装定義です。

ローカルモニタが排他アクセス状態で、プロセッサが最後に実行した排他ロードのアドレス以外のアドレスに対して排他ストアが実行される場合は、ストアによりメモリが更新されるかどうかは実装定義ですが、ローカルモニタは必ずオープンアクセス状態にリセットされます。この機構には、次の条件が適用されます。

- コンテキストスイッチで使用されます。*コンテキストスイッチのサポート* P. A3- 21 を参照して下さい。
- 他のすべての場合には、ソフトウェアプログラミング エラーとして扱う必要があります。

注

物理アドレスへのタグ付けを引き起こしたオブザーバとは別のオブザーバが、タグ付けされた物理アドレスへストアを行う場合、そのストアによりローカルモニタ内のタグがクリアされるかどうかは予測不能です。

ローカルモニタのステートマシンを、図 A3- 2 に示します。この図に示されている各操作の影響を、P. A3- 15 の表 A3- 6 に示します。

*マークにある演算は、使用可能な代替実装定義オプションを示します。

図で: LoadExclは任意の排他ロード命令を示します。
StoreExclは任意の排他ストア命令を示します。
Storeは他の任意のストア命令を示します。

どのLoadExcl演算も、タグ付きのアドレスを演算で使用されているアドレスxの最上位ビットに更新します。
詳細については、「タグが付けられたメモリブロックのサイズ」セクションを参照して下さい。

図 A3- 2 ローカルモニタのステートマシン図

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

For the local monitor state machine, as shown in Figure A3-2 on page A3-14:

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any physical address, but instead treats any access as matching the address of the previous `LoadExcl`.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive operations from other processors.
- It is UNPREDICTABLE whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

Table A3-6 shows the effect of the operations shown in Figure A3-2 on page A3-14.

Table A3-6 Effect of Exclusive instructions and write operations on local monitor

Initial state	Operation[a]	Effect	Final state
Open Access	`CLREX`	No effect	Open Access
Open Access	`StoreExcl(x)`	Does not update memory, returns status 1	Open Access
Open Access	`LoadExcl(x)`	Loads value from memory, tags address x	Exclusive Access
Open Access	`Store(x)`	Updates memory, no effect on monitor	Open Access
Exclusive Access	`CLREX`	Clears tagged address	Open Access
Exclusive Access	`StoreExcl(t)`	Updates memory, returns status 0	Open Access
Exclusive Access	`StoreExcl(!t)`	Updates memory, returns status 0[b]	Open Access
		Does not update memory, returns status 1[b]	
Exclusive Access	`LoadExcl(x)`	Loads value from memory, changes tag to address to x	Exclusive Access
Exclusive Access	`Store(!t)`	Updates memory, no effect on monitor	Exclusive Access
Exclusive Access	`Store(t)`	Updates memory	Exclusive Access[b]
			Open Access[b]

a. In the table:
`LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any store operation other than a Store-Exclusive operation.
t is the tagged address, bits [31:a] of the address of the last Load-Exclusive instruction. For more information, see *Tagging and the size of the tagged memory block* on page A3-20.

b. IMPLEMENTATION DEFINED alternative actions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

P. A3-14 の図 A3-2 に示すように、ローカルモニタのステートマシンには次の性質があります。

- ローカルモニタの実装定義オプションは、構築されたローカルモニタと整合性があります。ローカルモニタは物理アドレスを保持しませんが、代わりに以前の `LoadExcl` のアドレスの一致としてアクセスが取り扱われるためです。
- ローカルモニタの実装では、他のプロセッサからの排他ロード / 排他ストア操作を認識する必要はありません。
- `Store` または `StoreExcl` が別のオブザーバから実行される場合、排他アクセスからオープンアクセス状態への遷移が発生するかどうかは予測不能です。

P. A3-14 の図 A3-2 に示されている操作の影響を、表 A3-6 に示します。

表 A3-6 ローカルモニタ上での排他命令および書き込み動作の影響

初期状態	操作[a]	影響	最終状態
オープンアクセス	`CLREX`	影響なし	オープンアクセス
オープンアクセス	`StoreExcl(x)`	メモリを更新せずに、ステータス 1 を返す	オープンアクセス
オープンアクセス	`LoadExcl(x)`	メモリから値をロードし、アドレス x にタグを付ける	排他アクセス
オープンアクセス	`Store(x)`	メモリを更新するが、モニタには影響なし	オープンアクセス
排他アクセス	`CLREX`	タグを付けられたアドレスをクリアする	オープンアクセス
排他アクセス	`StoreExcl(t)`	メモリを更新して、ステータス 0 を返す	オープンアクセス
排他アクセス	`StoreExcl(!t)`	メモリを更新して、ステータス 0 を返す[b]	オープンアクセス
		メモリを更新せずに、ステータス 1 を返す[b]	
排他アクセス	`LoadExcl(x)`	メモリから値をロードして、アドレスへのタグを x に変更する	排他アクセス
排他アクセス	`Store(!t)`	メモリを更新するが、モニタには影響なし	排他アクセス
排他アクセス	`Store(t)`	メモリを更新する	排他アクセス[b]
			オープンアクセス[b]

a. この表の表記は次の意味です。
`LoadExcl` は、任意の排他ロード命令を表します。
`StoreExcl` は、任意の排他ストア命令を表します。
`Store` は、排他ストア以外の任意のストア操作を表します。
t はタグ付けされたアドレスで、最後の排他ロード命令のアドレスのビット [31:a] を表します。詳細については、*タグ付けとタグ付けされたメモリブロックのサイズ* P. A3-20 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.4.2 Exclusive access instructions and Shareable memory regions

For memory regions that have the *Shareable* attribute, exclusive access instructions rely on:

- A *local monitor* for each processor in the system, that tags any address from which the processor executes a Load-Exclusive. The local monitor operates as described in *Exclusive access instructions and Non-shareable memory regions* on page A3-13, except that for Shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of:
  - — updating memory
  - — returning a status value of 0.

  The local monitor can ignore exclusive accesses from other processors in the system.
- A *global monitor* that tags a physical address as exclusive access for a particular processor. This tag is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the tagged address by any other observer in the shareability domain of the memory location is guaranteed to clear the tag. For each processor in the system, the global monitor:
  - — holds a single tagged address
  - — maintains a state machine.

The global monitor can either reside in a processor block or exist as a secondary monitor at the memory interfaces.

Note

An implementation can combine the functionality of the global and local monitors into a single unit.

Operation of the global monitor

Load-Exclusive from *Shareable* memory performs a load from memory, and causes the physical address of the access to be tagged as exclusive access for the requesting processor. This access also causes the exclusive access tag to be removed from any other physical address that has been tagged by the requesting processor. The global monitor only supports a single outstanding exclusive access to Shareable memory per processor.

Store-Exclusive performs a conditional store to memory:

- The store is guaranteed to succeed only if the physical address accessed is tagged as exclusive access for the requesting processor and both the local monitor and the global monitor state machines for the requesting processor are in the Exclusive Access state. In this case:
  - — a status value of 0 is returned to a register to acknowledge the successful store
  - — the final state of the global monitor state machine for the requesting processor is IMPLEMENTATION DEFINED
  - — if the address accessed is tagged for exclusive access in the global monitor state machine for any other processor then that state machine transitions to Open Access state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

b. 実装定義の代替動作

A3.4.2 排他アクセス命令と共有可能メモリ領域

*共有可能*属性を持つメモリ領域の場合、排他アクセス命令は次の項目に依存します。

- システム内の各プロセッサに対する *ローカルモニタ*。そのプロセッサが排他ロードを実行するアドレスにタグ付けします。ローカルモニタは、*排他アクセス命令と共有不可メモリ領域* P. A3-13 に記載されているように動作します。ただし、共有可能メモリの場合、次の 1 つ以上を行うとそのセクションに記載されている排他ストア命令はすべて、グローバルモニタによるチェックを受けます。
  - – メモリの更新
  - – ステータス値 0 を返す

  ローカルモニタは、システム内の他のプロセッサからの排他アクセスを無視できます。
- *グローバルモニタ*。特定のプロセッサによる排他アクセス用に、物理アドレスにタグ付けします。このタグは以後、ローカルモニタによって失敗とされていないそのアドレスに対して排他ストアが発生しうるかどうかの判定に使用されます。メモリ位置の共有可能ドメインにおいて、他のオブザーバがタグの付いているアドレスに書き込みを行い、成功した場合、タグは必ずクリアされます。システム内の各プロセッサについて、グローバルモニタは次の処理を実行します。
  - – タグ付けされた単一アドレスの保持
  - – ステートマシンの維持

グローバルモニタは、あるプロセッサブロック内にあるか、またはメモリインタフェースの 2 次モニタとして存在します。

注

実装では、グローバルモニタとローカルモニタの機能を単一のユニットに結合することができます。

グローバルモニタの動作

*共有可能*メモリからの排他ロードを実行した場合、メモリからのロードが行われるとともに、アクセスされた物理アドレスには、要求を発行したプロセッサによる排他アクセス用のタグが付けられます。同時に、要求を発行したプロセッサによって他の物理アドレスが排他アクセスにタグ付けされていた場合、このアクセスによってそのタグは削除されます。グローバルモニタでは、共有可能メモリへの継続中の排他アクセスは、プロセッサごとに 1 つだけサポートされています。

排他ストアは、メモリへの条件付きストアを次のように実行します。

- アクセスされる物理アドレスに、要求を発行したプロセッサに対する排他アクセスのタグが付けられており、ローカルモニタおよびグローバルモニタのステートマシンが両方とも、そのプロセッサについて排他アクセス状態の場合にのみ、ストアの成功が保証されます。この場合、次の動作が行われます。
  - – ステータス値 0 がレジスタに返され、ストアの成功を知らせます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- If no address is tagged as exclusive access for the requesting processor, the store does not succeed:
  - a status value of 1 is returned to a register to indicate that the store failed
  - the global monitor is not affected and remains in Open Access state for the requesting processor.
- If a different physical address is tagged as exclusive access for the requesting processor, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
  - if the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned
  - if the global monitor state machine for the processor was in the Exclusive Access state before the Store-Exclusive it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each processor in the system. The state machine for accesses to Shareable memory by processor (n) can respond to all the Shareable memory accesses visible to it. This means it responds to:

- accesses generated by the associated processor (n)
- accesses generated by the other observers in the shareability domain of the memory location (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive in the system.

Figure A3-3 on page A3-18 shows the state machine for processor(n) in a global monitor. Table A3-7 on page A3-19 shows the effect of each of the operations shown in the figure.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

  - 要求を発行したプロセッサに対するグローバルモニタのステートマシンの最終状態は、実装定義です。
  - アクセスされるアドレスが、グローバルモニタのステートマシン内で、他のプロセッサに対する排他アクセスとしてタグ付けされている場合、そのステートマシンはオープンアクセス状態へ遷移します。
- 要求を発行したプロセッサに対して排他アクセスとしてタグ付けされているアドレスがない場合、ストアは成功せず、次の動作が行われます。
  - ステータス値 1 がレジスタに返され、ストアが失敗したことを示します。
  - グローバルモニタは影響を受けず、要求を発行したプロセッサに対してオープンアクセス状態が維持されます。
- 要求を発行したプロセッサについて、別の物理アドレスが排他アクセスにタグ付けされている場合、ストアが成功するかどうかは実装定義です。
  - ストアが成功した場合はステータス値 0 が、失敗した場合は値 1 がレジスタに返されます。
  - 排他ストアが行われる前の時点で、そのプロセッサに対するグローバルモニタのステートマシンが排他アクセス状態だった場合、そのステートマシンがオープンアクセス状態に遷移するかどうかは実装定義です。

ステータス値が返されるレジスタは、排他ストア命令によって定義されます。

共有メモリシステムでは、グローバルモニタには、システム内の各プロセッサに対して個別のステートマシンが実装されます。プロセッサ (n) による共有可能メモリへのアクセスに対するステートマシンは、参照可能なすべての共有可能メモリアクセスに応答することができます。これは、次のアクセスに応答することを意味します。

- 対応するプロセッサ (n) によって生成されるアクセス
- メモリ位置の共有可能ドメイン (!n) において他のオブザーバによって生成されるアクセス

共有メモリシステムでは、グローバルモニタは、システム中の排他ロードまたは排他ストアを生成する可能性のあるオブザーバごとに別々のステートマシンを実装します。

グローバルモニタのプロセッサ (n) に対するステートマシンを、P. A3-18 の図 A3-3 に示します。この図に示されている各操作の影響を、P. A3-19 の表 A3-7 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

‡ `StoreExcl`(Tagged_Address,!n) clears the monitor only if the `StoreExcl` updates memory

Operations marked * are possible alternative IMPLEMENTATION DEFINED options.

In the diagram: `LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any other store instruction.

Any `LoadExcl` operation updates the tagged address to the most significant bits of the address x used for the operation. For more information see the section *Size of the tagged memory block*.

Figure A3-3 Global monitor state machine diagram for processor(n) in a multiprocessor system

Note

For the global monitor state machine, as shown in Figure A3-3:

- Whether a Store-Exclusive successfully updates memory or not depends on whether the address accessed matches the tagged Shareable memory address for the processor issuing the Store-Exclusive instruction. For this reason, Figure A3-3 and Table A3-7 on page A3-19 only show how the (!n) entries cause state transitions of the state machine for processor(n).
- An Load-Exclusive can only update the tagged Shareable memory address for the processor issuing the Load-Exclusive instruction.
- The effect of the `CLREX` instruction on the global monitor is IMPLEMENTATION DEFINED.
- It is IMPLEMENTATION DEFINED:
  - — whether a modification to a non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state
  - — whether a Load-Exclusive to a non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

‡ StoreExclがメモリを更新する場合にのみ、StoreExcl(Tagged_Address,!n) がモニタをクリアします。

*マークにある演算は、使用可能な代替実装定義オプションを示します。

図で: LoadExclは任意の排他ロード命令を示します。
StoreExclは任意の排他ストア命令を示します。
Storeは他の任意のストア命令を示します。

どのLoadExcl演算も、タグ付きのアドレスを演算で使用されているアドレスxの最上位ビットに更新します。
詳細については、「タグが付けられたメモリブロックのサイズ」セクションを参照して下さい。

図 A3-3 マルチプロセッサシステムにおけるプロセッサ(n)に対するグローバルモニタのステートマシン図

注

図 A3-3 に示されているように、グローバルモニタのステートマシンには次の性質があります。

- 排他ストアによりメモリが正常に更新されるかどうかは、アクセスされたアドレスが、その排他ストア命令を発行するプロセッサに対してタグ付けされた共用可能なメモリアドレスと、一致するかどうかによって決まります。この理由のため、(!n) エントリがプロセッサ (n) のステートマシンの状態遷移を発生させる方法のみを図 A3-3 および P. A3-19 の表 A3-7 に示します。
- 排他ロードでは、その排他ロード命令を発行するプロセッサに対してタグ付けされている共有可能のメモリアドレスのみを更新できます。
- グローバルモニタの CLREX 命令の作用は実装定義です。
- 次の条件は実装定義です。
  - — 共有不可メモリ位置に対する変更が、排他アクセス状態からオープンアクセス状態へのグローバルモニタの遷移を引き起こすかどうか
  - — 共有不可メモリ位置に対する排他ロードが、オープンアクセス状態から排他アクセス状態へのグローバルモニタの遷移を引き起こすかどうか

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A3-7 shows the effect of the operations shown in Figure A3-3 on page A3-18.

Table A3-7 Effect of load/store operations on global monitor for processor(n)

Initial state[a]	Operation[b]	Effect	Final state[a]
Open	CLREX(n), CLREX(!n)	None	Open
	StoreExcl(x,n)	Does not update memory, returns status 1	Open
	LoadExcl(x,!n)	Loads value from memory, no effect on tag address for processor(n)	Open
	StoreExcl(x,!n)	Depends on state machine and tag address for processor issuing STREX[c]	Open
	Store(x,n), Store(x,!n)	Updates memory, no effect on monitor	Open
	LoadExcl(x,n)	Loads value from memory, tags address x	Exclusive
Exclusive	LoadExcl(x,n)	Loads value from memory, tags address x	Exclusive
	CLREX(n)	None. Effect on the final state is IMPLEMENTATION DEFINED.	Exclusive[e]
			Open[e]
	CLREX(!n)	None	Exclusive
	StoreExcl(t,!n)	Updates memory, returns status 0[c]	Open
		Does not update memory, returns status 1[c]	Exclusive
	StoreExcl(t,n)	Updates memory, returns status 0[d]	Open
			Exclusive
	StoreExcl(!t,n)	Updates memory, returns status 0[e]	Open
			Exclusive
		Does not update memory, returns status 1[e]	Open
			Exclusive
	StoreExcl(!t,!n)	Depends on state machine and tag address for processor issuing STREX	Exclusive

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

P. A3-18 の図 A3-3 に示されている操作の影響を表 A3-7 に示します。

表 A3-7 プロセッサ (n) に対するグローバルモニタのロード / ストア操作の影響

初期状態[a]	操作[b]	影響	最終状態[a]
オープン	CLREX(n)、CLREX(!n)	なし	オープン
	StoreExcl(x,n)	メモリを更新せずに、ステータス 1 を返す。	オープン
	LoadExcl(x,!n)	メモリから値をロードするが、プロセッサ (n) に対するタグアドレスには影響なし。	オープン
	StoreExcl(x,!n)	STREX[c] を発行するプロセッサに対するステートマシンおよびタグアドレスに依存	オープン
	Store(x,n)、Store(x,!n)	メモリを更新するが、モニタには影響なし。	オープン
	LoadExcl(x,n)	メモリから値をロードし、アドレス x にタグを付ける。	排他
排他	LoadExcl(x,n)	メモリから値をロードし、アドレス x にタグを付ける。	排他
	CLREX(n)	なし。最終状態は実装定義です。	排他[e]
			オープン[e]
	CLREX(!n)	なし	排他
	StoreExcl(t,!n)	メモリを更新して、ステータス 0 を返す。[c]	オープン
		メモリを更新せずに、ステータス 1 を返す。[c]	排他
	StoreExcl(t,n)	メモリを更新して、ステータス 0 を返す。[d]	オープン
			排他
	StoreExcl(!t,n)	メモリを更新して、ステータス 0 を返す。[e]	オープン
			排他
		メモリを更新せずに、ステータス 1 を返す。[e]	オープン
			排他
	StoreExcl(!t,!n)	STREX を発行するプロセッサに対するステートマシンおよびタグアドレスに依存	排他

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A3-7 Effect of load/store operations on global monitor for processor(n) (continued)

Initial state[a]	Operation[b]	Effect	Final state[a]
Exclusive	`Store(t,n)`	Updates memory	Exclusive[e]
			Open[e]
	`Store(t,!n)`	Updates memory	Open
	`Store(!t,n)`, `Store(!t,!n)`	Updates memory, no effect on monitor	Exclusive

a. Open = Open Access state, Exclusive = Exclusive Access state.
b. In the table:
`LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any store operation other than a Store-Exclusive operation.
t is the tagged address for processor(n), bits [31:a] of the address of the last Load-Exclusive instruction issued by processor(n), see *Tagging and the size of the tagged memory block*.
c. The result of a `STREX(x,!n)` or a `STREX(t,!n)` operation depends on the state machine and tagged address for the processor issuing the `STREX` instruction. This table shows how each possible outcome affects the state machine for processor(n).
d. After a successful `STREX` to the tagged address, the state of the state machine is IMPLEMENTATION DEFINED. However, this state has no effect on the subsequent operation of the global monitor.
e. Effect is IMPLEMENTATION DEFINED. The table shows all permitted implementations.

A3.4.3 Tagging and the size of the tagged memory block

As stated in the footnotes to Table A3-6 on page A3-15 and Table A3-7 on page A3-19, when a Load-Exclusive instruction is executed, the resulting tag address ignores the least significant bits of the memory address.

```
Tagged_address = Memory_address[31:a]
```

The value of a in this assignment is IMPLEMENTATION DEFINED, between a minimum value of 3 and a maximum value of 11. For example, in an implementation where a == 4, a successful `LDREX` of address `0x000341B4` gives a tag value of bits [31:4] of the address, giving `0x000341B`. This means that the four words of memory from `0x000341B0` to `0x000341BF` are tagged for exclusive access.

The size of the tagged memory block called the *Exclusives Reservation Granule*. The Exclusives Reservation Granule is IMPLEMENTATION DEFINED between:

- two words, in an implementation with a == 3
- 512 words, in an implementation with a == 11.

In some implementations the CTR identifies the Exclusives Reservation Granule, see:

- *c0, Cache Type Register (CTR)* on page B3-83 for a VMSA implementation
- *c0, Cache Type Register (CTR)* on page B4-34 for a PMSA implementation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A3-7 プロセッサ (n) に対するグローバルモニタのロード / ストア操作の影響（続き）

初期状態[a]	操作[b]	影響	最終状態[a]
排他	`Store(t,n)`	メモリを更新する。	排他[e]
			オープン[e]
	`Store(t,!n)`	メモリを更新する。	オープン
	`Store(!t,n)`、`Store(!t,!n)`	メモリを更新するが、モニタには影響なし。	排他

a. オープン = オープンアクセス状態、排他 = 排他アクセス状態。
b. この表の表記は次の意味です。
`LoadExcl` は、任意の排他ロード命令を表します。
`StoreExcl` は、任意の排他ストア命令を表します。
`Store` は、排他ストア以外の任意のストア操作を表します。
t はプロセッサ (n) に対してタグ付けされたアドレスで、プロセッサ (n) によって発行された最後の排他ロード命令のアドレスのビット [31:a] を表します。詳細については、*タグ付けとタグ付けされたメモリブロックのサイズ*を参照して下さい。
c. `STREX(x,!n)` または `STREX(t,!n)` 演算の結果は、`STREX` 命令を発行するプロセッサに対するステートマシン、およびタグ付けされたアドレスに関係します。この表では、可能な結果のそれぞれについて、プロセッサ (n) のステートマシンがどのような影響を受けるかを示しています。
d. タグ付けされたアドレスへの `STREX` が成功した場合、ステートマシンの状態は実装定義です。ただし、この状態はグローバルモニタの以後の操作には影響を与えません。
e. 作用は実装定義です。この表では、許可されている実装をすべて示しています。

A3.4.3 タグ付けとタグ付けされたメモリブロックのサイズ

P. A3-15 の表 A3-6 および P. A3-19 の表 A3-7 の脚注に記載されているように、排他ロード命令が実行されるとき、結果のタグアドレスではメモリアドレスの最下位ビットが無視されます。

```
Tagged_address = Memory_address[31:a]
```

この割り当てにおける a という値は、3 〜 11 の範囲で実装定義の値です。例えば、a == 4 という実装の場合、アドレス `0x000341B4` の `LDREX` 命令が成功すると、タグ値はアドレスのビット [31:4]、つまり `0x000341B` になります。これは、`0x000341B0` 〜 `0x000341BF` の 4 ワードのメモリに、排他アクセスのタグが付けられたことを意味します。

タグ付けされたメモリブロックのサイズは、*排他予約粒度*と呼ばれます。排他予約粒度は、次の範囲で実装定義です。

- 2 ワード。a == 3 の実装
- 512 ワード。a == 11 の実装

一部の実装では、CTR によって排他予約粒度が識別されます（以下参照）。

- *c0、キャッシュタイプ レジスタ (CTR)* P. B3-83（VMSA 実装の場合）
- *c0、キャッシュタイプ レジスタ (CTR)* P. B4-34 （PMSA 実装の場合）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.4.4 Context switch support

After a context switch, software must ensure that the local monitor is in the Open Access state. This requires it to either:

- execute a CLREX instruction
- execute a dummy STREX to a memory address allocated for this purpose.

Note

- Using a dummy STREX for this purpose is backwards-compatible with the ARMv6 implementation of the exclusive operations. The CLREX instruction is introduced in ARMv6K.
- Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

The STREX or CLREX instruction following a context switch might cause a subsequent Store-Exclusive to fail, requiring a load … store sequence to be replayed. To minimize the possibility of this happening, ARM recommends that the Store-Exclusive instruction is kept as close as possible to the associated Load-Exclusive instruction, see *Load-Exclusive and Store-Exclusive usage restrictions*.

A3.4.5 Load-Exclusive and Store-Exclusive usage restrictions

The Load-Exclusive and Store-Exclusive instructions are intended to work together, as a pair, for example a LDREX/STREX pair or a LDREXB/STREXB pair. As mentioned in *Context switch support*, ARM recommends that the Store-Exclusive instruction always follows within a few instructions of its associated Load-Exclusive instructions. To support different implementations of these functions, software must follow the notes and restrictions given here.

These notes describe use of an LDREX/STREX pair, but apply equally to any other Load-Exclusive/Store-Exclusive pair:

- The exclusives support a single outstanding exclusive access for each processor thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the IsExclusiveLocal() function. If the target address of an STREX is different from the preceding LDREX in the same execution thread, behavior can be UNPREDICTABLE. As a result, an LDREX/STREX pair can only be relied upon to eventually succeed if they are executed with the same address. Where a context switch or exception might result in a change of execution thread, a CLREX instruction or a dummy STREX instruction must be executed to avoid unwanted effects, as described in *Context switch support* Using an STREX in this way is the only occasion where software can program an STREX with a different address from the previously executed LDREX.
- An explicit store to memory can cause the clearing of exclusive monitors associated with other processors, therefore, performing a store between the LDREX and the STREX can result in a livelock situation. As a result, code must avoid placing an explicit store between an LDREX and an STREX in a single code sequence.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.4.4 コンテキストスイッチのサポート

コンテキストスイッチの後、ソフトウェアはローカルモニタがオープンアクセス状態にあることを保証する必要があります。このためには、次のいずれかを行う必要があります。

- CLREX 命令の実行
- この目的で割り当てられたメモリアドレスへのダミー STREX の実行

注

- この目的でダミー STREX を使用することは、排他演算の ARMv6 実装と下位互換です。ARMv6K では CLREX 命令が導入されています。
- コンテキストスイッチは、アプリケーションレベルの操作ではありません。ただし、この情報は排他操作の説明の一部としてここで紹介されています。

コンテキストスイッチの後で STREX または CLREX 命令を使用した場合、その後に実行される排他ストアが失敗する可能性があり、その場合にはロード ... ストアのシーケンスを再実行する必要があります。この条件が発生する可能性を最小限にするため、排他ストア命令は関連する排他ロード命令のできるだけ近くに配置することをお勧めします。詳細については、*排他ロード / 排他ストアの使用に関する制約事項*を参照して下さい。

A3.4.5 排他ロード / 排他ストアの使用に関する制約事項

排他ロード / 排他ストア命令は、1 つのペアとして動作するように設計されています。例として、LDREX/STREX のペアや LDREXB/STREXB のペアが挙げられます。*コンテキストスイッチのサポート*で説明されているように、排他ストア命令は常に関連する排他ロード命令から数命令以内に配置することをお勧めします。これらの関数について各種の実装をサポートするため、ここで示されている注意事項および制約事項に従うようにソフトウェアを設計する必要があります。

これらの注意事項では LDREX/STREX のペアの使用法について説明されていますが、同様に他の排他ロード / 排他ストアのペアにも適用されます。

- 排他命令は、実行される各プロセッサスレッドごとに、継続中の排他アクセスを 1 つサポートします。アーキテクチャはこの排他アクセスを使用することで、IsExclusiveLocal() 関数の一部としてアドレスおよびサイズのチェックを行う必要がなくなります。同じ実行スレッド内で STREX のターゲットアドレスが前の LDREX のアドレスと異なる場合、動作は予測不能です。結果的に、LDREX/STREX のペアは同じアドレスで実行されている場合のみ最終的に成功することが保証されます。コンテキストスイッチや例外が実行スレッドの変更を引き起こす場合は、CLREX 命令またはダミー STREX 命令を実行し、*コンテキストスイッチのサポート*に説明されているような望ましくない影響を防ぐ必要があります。STREX をこのような方法に使う場合のみ、ソフトウェアにおいて、前に実行した LDREX と異なるアドレスで STREX をプログラムできます。
- メモリへの明示的なストアにより他のプロセッサに関連する排他モニタのクリアが実行される可能性があるため、LDREX と STREX との間でのストアの実行はライブロック状況を引き起こす場合があります。このため、コーディングにおいて、単一のコードシーケンスで LDREX と STREX との間に明示的なストアを置くことは避ける必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- If two STREX instructions are executed without an intervening LDREX the second STREX returns a status value of 1. This means that:
  - every STREX must have a preceding LDREX associated with it in a given thread of execution
  - it is not necessary for every LDREX to have a subsequent STREX.
- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a Store-Exclusive is the same as the transaction size of the preceding Load-Exclusive that was executed in that thread. If the transaction size of a Store-Exclusive is different from the preceding Load-Exclusive in the same execution thread, behavior can be UNPREDICTABLE. As a result, an LDREX/STREX pair can only be relied upon to eventually succeed only if they have the same size. Where a context switch or exception might result in a change of execution thread, the software must execute a CLREX instruction or a dummy STREX instruction to avoid unwanted effects, as described in *Context switch support* on page A3-21. Using an STREX in this way is the only occasion where software can use a Store-Exclusive instruction with a different transaction size from the previously executed Load-Exclusive instruction.
- An implementation might clear an exclusive monitor between the LDREX and the STREX, without any application-related cause. For example, this might happen because of cache evictions. Code written for such an implementation must avoid having any explicit memory accesses or cache maintenance operations between the LDREX and STREX instructions.
- Implementations can benefit from keeping the LDREX and STREX operations close together in a single code sequence. This minimizes the likelihood of the exclusive monitor state being cleared between the LDREX instruction and the STREX instruction. Therefore, ARM strongly recommends a limit of 128 bytes between LDREX and STREX instructions in a single code sequence, for best performance.
- Implementations that implement coherent protocols, or have only a single master, might combine the local and global monitors for a given processor. The IMPLEMENTATION DEFINED and UNPREDICTABLE parts of the definitions in *Exclusive monitors operations* on page B2-35 are provided to cover this behavior.
- The architecture sets an upper limit of 2048 bytes on the size of a region that can be marked as exclusive. Therefore, for performance reasons, ARM recommends that software separates objects that will be accessed by exclusive accesses by at least 2048 bytes. This is a performance guideline rather than a functional requirement.
- LDREX and STREX operations must be performed only on memory with the Normal memory attribute.
- The effect of Data Abort exceptions on the state of monitors is UNPREDICTABLE. ARM recommends that abort handling code performs a CLREX instruction or a dummy STREX instruction to clear the monitor state.
- If the memory attributes for the memory being accessed by an LDREX/STREX pair are changed between the LDREX and the STREX, behavior is UNPREDICTABLE.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- 2 つの STREX 命令を、間に LDREX 命令を入れないで実行すると、2 番目の STREX によりステータス値 1 が返されます。これは次のことを意味します。
  - ある実行スレッド中では、すべての STREX 命令の前には、その命令に対応する LDREX 命令が配置される必要があります。
  - LDREX 命令の後に、必ず STREX 命令を配置する必要はありません。
- 排他ロード / 排他ストア命令の実装によっては、いかなる実行スレッドにおいても、排他ストアのトランザクションサイズが、そのスレッドで実行された前の排他ロードのトランザクションサイズと同じであることが要求される可能性があります。同じ実行スレッド内で、排他ストアのトランザクションサイズが、前の排他ロードのトランザクションサイズと異なる場合、動作は予測不能です。結果的に、LDREX/STREX のペアは、双方が同じサイズの場合のみ、最終的に成功することが保証されます。コンテキストスイッチや例外によって実行スレッドが変更される可能性がある場合、*コンテキストスイッチのサポート* P. A3-21 に説明されているような望ましくない効果を回避するため、ソフトウェアで CLREX 命令またはダミー STREX 命令を実行する必要があります。STREX をこの方法で使用する場合に限り、ソフトウェアにおいて、前に実行した排他ロード命令とは異なるトランザクションサイズの排他ストア命令を使用できます。
- 実装によっては、アプリケーション関連の原因がなくても LDREX と STREX との間で排他モニタがクリアされる場合があります。例えば、キャッシュの退出によってクリアが発生する場合があります。そのような実装用に記述されたコードでは、LDREX 命令と STREX 命令との間で、明示的なメモリアクセスやキャッシュ保守操作を避ける必要があります。
- 実装によっては、単一のコードシーケンス内で LDREX 操作と STREX 操作をすぐ近くに配置することで利益が得られる場合があります。これにより、LDREX 命令と STREX 命令との間で排他モニタ状態がクリアされる可能性を最小限にすることができます。このため、最良の性能を得るために、単一のコードシーケンス内では LDREX 命令と STREX 命令の間が 128 バイト以内になるように制限を設けることを強くお勧めします。
- コヒーレントプロトコルを実装しているか、または単一のマスタのみが存在する実装では、あるプロセッサに対するローカルモニタとグローバルモニタとが結合されている場合があります。*排他モニタ操作* P. B2-35 の定義で実装定義および予測不能と記載されている部分は、このような動作に対応しています。
- アーキテクチャでは、排他としてマークされる領域サイズの上限は 2048 バイトに設定されます。そのため、性能上の理由から、ソフトウェアを作成するとき、排他アクセスによってアクセスされるオブジェクトは互いに少なくとも 2048 バイト離して配置することをお勧めします。これは、機能的な要件ではなく性能上のガイドラインです。
- LDREX 操作と STREX 操作は、ノーマルメモリ属性を持つメモリ上でのみ実行できます。
- モニタ状態におけるデータアボート例外の影響は予測不能です。アボート処理コードで、CLREX 命令またはダミー STREX 命令を実行してモニタ状態をクリアすることをお勧めします。
- LDREX/STREX のペアによってアクセスされるメモリのメモリ属性が LDREX と STREX との間で変更されると、動作は予測不能になります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.4.6 Semaphores

The Swap (SWP) and Swap Byte (SWPB) instructions must be used with care to ensure that expected behavior is observed. Two examples are as follows:

1. A system with multiple bus masters that uses Swap instructions to implement semaphores that control interactions between different bus masters.

   In this case, the semaphores must be placed in an uncached region of memory, where any buffering of writes occurs at a point common to all bus masters using the mechanism. The Swap instruction then causes a locked read-write bus transaction.

2. A systems with multiple threads running on a uniprocessor that uses the Swap instructions to implement semaphores that control interaction of the threads.

   In this case, the semaphores can be placed in a cached region of memory, and a locked read-write bus transaction might or might not occur. The Swap and Swap Byte instructions are likely to have better performance on such a system than they do on a system with multiple bus masters such as that described in example 1.

Note

From ARMv6, use of the Swap and Swap Byte instructions is deprecated. All new software should use the Load-Exclusive and Store-Exclusive synchronization primitives described in *Synchronization and semaphores* on page A3-12, for example LDREX and STREX.

A3.4.7 Synchronization primitives and the memory order model

The synchronization primitives follow the memory order model of the memory type accessed by the instructions. For this reason:

- Portable code for claiming a spin-lock must include a *Data Memory Barrier* (DMB) operation, performed by a DMB instruction, between claiming the spin-lock and making any access that makes use of the spin-lock.
- Portable code for releasing a spin-lock must include a DMB instruction before writing to clear the spin-lock.

This requirement applies to code using:

- the Load-Exclusive/Store-Exclusive instruction pairs, for example LDREX/STREX
- the deprecated synchronization primitives, SWP/SWPB.

A3.4.8 Use of WFE and SEV instructions by spin-locks

ARMv7 and ARMv6K provide Wait For Event and Send Event instructions, WFE and SEV, that can assist with reducing power consumption and bus contention caused by processors repeatedly attempting to obtain a spin-lock. These instructions can be used at application level, but a complete understanding of what they do depends on system-level understanding of exceptions. They are described in *Wait For Event and Send Event* on page B1-44.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.4.6 セマフォ

スワップ (SWP) およびバイトスワップ (SWPB) 命令は、期待どおりの動作が観測されるよう慎重に使用する必要があります。次に 2 つの例を挙げます。

1. 複数のバスマスタを持つシステムが、スワップ命令を使用し、異なるバスマスタ間の相互動作を制御するセマフォを実装する場合。

   この場合、セマフォはメモリの非キャッシュ領域に、この機構を使用するすべてのバスマスタに対して書き込みのバッファが共通の地点で発生するような場所に配置する必要があります。この場合、スワップ命令により、ロックされた読み出し - 書き込みバストランザクションが引き起こされます。

2. 複数のスレッドをユニプロセッサ上で実行するシステムが、スワップ命令を使用し、スレッドの相互動作を制御するセマフォを実装する場合。

   この場合、セマフォはメモリのキャッシュ領域に配置することができ、ロックされた読み出し - 書き込みバストランザクションが発生することも発生しないこともあります。例 1 で説明したような、複数のバスマスタを持つシステムよりも、このようなシステムのほうが、スワップ命令とバイトスワップ命令の性能が高くなる傾向にあります。

注

ARMv6 およびそれ以降では、スワップおよびバイトスワップ命令の使用は推奨されません。新しいソフトウェアはすべて、*同期化とセマフォ* P. A3-12 で説明されている LDREX および STREX など、排他ロード / 排他ストア同期化基本命令を使用することが推奨されます。

A3.4.7 同期化基本命令とメモリオーダ モデル

同期化基本命令は、その命令によってアクセスされるメモリタイプのメモリオーダ モデルに従います。この理由のため、次の処理が必要になります。

- スピンロックを要求する移植可能なコードでは、スピンロックの要求とスピンロックを使用するアクセス実行との間に、DMB 命令により実行される *データ メモリバリア*(DMB) 動作が含まれている必要があります。
- スピンロックを解放する移植可能なコードでは、スピンロックをクリアする書き込みの前に DMB 命令が含まれている必要があります。

この要件は、次のものを使用するコードに適用されます。

- LDREX/STREX など、排他ロード / 排他ストア命令のペア
- SWP/SWPB など、推奨されない同期化基本命令

A3.4.8 スピンロックによる WFE、SEV 命令の使用

ARMv7 と ARMv6K には、イベント待ち命令の WFE とイベント送信命令の SEV 命令があります。これらは消費電力を削減し、プロセッサが繰り返しスピンロックを取得しようとすることによるバスの競合を減らします。これらの命令はアプリケーションレベルで使用できますが、どのような動作をするかを完全に理解するには、例外に関するシステムレベルでの知識が必要です。詳細については、*イベント待ちとイベント送信* P. B1-44 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.5 Memory types and attributes and the memory order model

ARMv6 defined a set of memory attributes with the characteristics required to support the memory and devices in the system memory map. In ARMv7 this set of attributes is extended by the addition of the Outer Shareable attribute for Normal memory.

Note

Whether an ARMv7 implementation supports the Outer Shareable memory attribute is IMPLEMENTATION DEFINED.

The ordering of accesses for regions of memory, referred to as the memory order model, is defined by the memory attributes. This model is described in the following sections:

- *Memory types*
- *Summary of ARMv7 memory attributes* on page A3-25
- *Atomicity in the ARM architecture* on page A3-26
- *Normal memory* on page A3-28
- *Device memory* on page A3-33
- *Strongly-ordered memory* on page A3-34
- *Memory access restrictions* on page A3-35
- *Backwards compatibility* on page A3-37
- *The effect of the Security Extensions* on page A3-37.

A3.5.1 Memory types

For each memory region, the most significant memory attribute specifies the memory type. There are three mutually exclusive memory types:

- Normal
- Device
- Strongly-ordered.

Normal and Device memory regions have additional attributes.

Usually, memory used for program code and for data storage is Normal memory. Examples of Normal memory technologies are:

- programmed Flash ROM

  Note

  During programming, Flash memory can be ordered more strictly than Normal memory.

- ROM
- SRAM
- DRAM and DDR memory.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A3.5 メモリのタイプおよび属性とメモリオーダ モデル

ARMv6 では、システム メモリマップ内でメモリおよびデバイスをサポートするために必要な特性を持つ、メモリ属性のセットが定義されています。ARMv7 では、この属性のセットにノーマルメモリ用の外部共有可能属性が追加されています。

注

ARMv7 の実装が外部共有可能メモリ属性をサポートするかどうかは実装定義です。

メモリオーダ モデルと呼ばれるメモリ領域に対するアクセスの順序付けは、メモリ属性によって定義されます。このモデルについては、次のセクションで説明します。

- *メモリタイプ*
- *ARMv7 のメモリ属性の要約* P. A3-25
- *ARM アーキテクチャにおけるアトミック性* P. A3-26
- *ノーマルメモリ* P. A3-29
- *デバイスメモリ* P. A3-33
- *ストロングリオーダ メモリ* P. A3-34
- *メモリアクセスの制約事項* P. A3-35
- *下位互換性* P. A3-37
- *セキュリティ拡張機能の効果* P. A3-37

A3.5.1 メモリタイプ

各メモリ領域に対して、最も重要なメモリ属性がメモリタイプを決定します。メモリタイプには次の 3 つが存在し、これらは相互に排他です。

- ノーマル
- デバイス
- ストロングリオーダ

ノーマルおよびデバイスのメモリ領域には、追加の属性が存在します。

一般に、プログラムコードおよびデータ記憶域に使用されるメモリはノーマルメモリです。ノーマルメモリテクノロジの例には、次のものがあります。

- プログラムされたフラッシュ ROM

  注

  プログラミング中のフラッシュメモリは、ノーマルメモリよりさらに順序付け規則が厳しくなることがあります。

- ROM
- SRAM
- DRAM および DDR メモリ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

System peripherals (I/O) generally conform to different access rules to Normal memory. Examples of I/O accesses are:

- FIFOs where consecutive accesses
  - add queued values on write accesses
  - remove queued values on read accesses.
- interrupt controller registers where an access can be used as an interrupt acknowledge, changing the state of the controller itself
- memory controller configuration registers that are used to set up the timing and correctness of areas of Normal memory
- memory-mapped peripherals, where accessing a memory location can cause side effects in the system.

In ARMv7, regions of the memory map for these accesses are defined as Device or Strongly-ordered memory. To ensure system correctness, access rules for Device and Strongly-ordered memory are more restrictive than those for Normal memory:

- both read and write accesses can have side effects
- accesses must not be repeated, for example, on return from an exception
- the number, order and sizes of the accesses must be maintained.

In addition, for Strongly-ordered memory, all memory accesses are strictly ordered to correspond to the program order of the memory access instructions.

A3.5.2 Summary of ARMv7 memory attributes

Table A3-8 summarizes the memory attributes. For more information about theses attributes see:

- *Normal memory* on page A3-28 and *Shareable attribute for Device memory regions* on page A3-34, for the *shareability* attribute
- *Write-Through Cacheable, Write-Back Cacheable and Non-cacheable Normal memory* on page A3-32, for the *cacheability* attribute.

Table A3-8 Memory attribute summary

Memory type attribute	Shareability	Other attributes	Description
Strongly-ordered	-	-	All memory accesses to Strongly-ordered memory occur in program order. All Strongly-ordered regions are assumed to be Shareable.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

システムペリフェラル (I/O) は、一般にノーマルメモリとは別のアクセス規則に準拠します。I/O アクセスの例には、次のものがあります。

- 連続アクセスにより次の処理が行われる FIFO
  - 書き込みアクセスで、キューに値を追加する。
  - 読み出しアクセスで、キューから値を削除する。
- 割り込みコントローラレジスタ。アクセスが割り込み応答として使用され、コントローラ自身の状態を変化させます。
- ノーマルメモリ領域のタイミングおよび正確さの設定に使用されるメモリコントローラ構成レジスタ
- メモリマップされたペリフェラルで、メモリ位置へのアクセスにより、システム内に副作用が引き起こされる場合

ARMv7 では、これらのアクセス用のメモリマップ領域は、デバイスまたはストロングリオーダメモリとして定義されます。システムの正確さを保証するために、デバイスおよびストロングリオーダメモリのアクセス規則は、ノーマルメモリの規則よりも厳格になっています。

- 読み出し / 書き込みアクセスのいずれにも、副作用の可能性があります。
- 例外からの復帰時などに、アクセスを繰り返し行うことはできません。
- アクセスの数、順序、サイズを維持する必要があります。

さらに、ストロングリオーダ メモリの場合は、すべてのメモリアクセスは厳密に順序付けられており、メモリアクセス命令のプログラム上の順序に対応しています。

A3.5.2 ARMv7 のメモリ属性の要約

メモリ属性の要約を、表 A3-8 に示します。これらの属性の詳細については、次のセクションを参照して下さい。

- *ノーマルメモリ* P. A3-29 および*デバイスメモリ領域に対する共有能可属性* P. A3-34（*共有可能*属性の場合）
- *ライトスルーキャッシュ可、ライトバックキャッシュ可、およびキャッシュ不可のノーマルメモリ* P. A3-32（*キャッシュ可*属性の場合）

表 A3-8 メモリ属性の要約

メモリタイプの属性	共有可能性	その他の属性	説明
ストロングリオーダ	–	–	ストロングリオーダ メモリに対するメモリアクセスは、すべてプログラム順に発生します。ストロングリオーダ領域は、すべて共有可能であることを前提としています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A3-8 Memory attribute summary (continued)

Memory type attribute	Shareability	Other attributes	Description
Device	Shareable	-	Intended to handle memory-mapped peripherals that are shared by several processors.
	Non-shareable	-	Intended to handle memory-mapped peripherals that are used only by a single processor.
Normal	Outer Shareable	Cacheability, one of: [a] Non-cacheable Write-Through Cacheable Write-Back Write-Allocate Cacheable Write-Back no Write-Allocate Cacheable	The Outer Shareable attribute qualifies the Shareable attribute for Normal memory regions and enables two levels of Normal memory sharing.[b]
	Inner Shareable	Cacheability, one of: [a] Non-cacheable Write-Through Cacheable Write-Back Write-Allocate Cacheable Write-Back no Write-Allocate Cacheable	Intended to handle Normal memory that is shared between several processors.
	Non-shareable	Cacheability, one of: [a] Non-cacheable Write-Through Cacheable Write-Back Write-Allocate Cacheable Write-Back no Write-Allocate Cacheable	Intended to handle Normal memory that is used by only a single processor.

a. The cacheability attribute is defined independently for inner and outer cache regions.
b. The significance of the Outer Shareable attribute is IMPLEMENTATION DEFINED.

A3.5.3 Atomicity in the ARM architecture

Atomicity is a feature of memory accesses, described as *atomic* accesses. The ARM architecture description refers to two types of atomicity, defined in:

- *Single-copy atomicity* on page A3-27
- *Multi-copy atomicity* on page A3-28.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A3-8 メモリ属性の要約（続き）

メモリタイプの属性	共有可能性	その他の属性	説明
デバイス	共有可能	−	複数のプロセッサにより共有されるメモリマップされたペリフェラルを処理することを意図しています。
	共有不可	−	単一プロセッサによってのみ使用されるメモリマップされたペリフェラルを処理することを意図しています。
ノーマル	外部共有可能	キャッシュ可。次のうちのいずれかです。[a] キャッシュ不可 ライトスルーキャッシュ可 ライトバック、書き込み割り当てキャッシュ可 ライトバック、書き込み割り当てなしキャッシュ可	外部共有可能属性は、ノーマルメモリ領域の共有可能属性を修飾し、2つのレベルのノーマルメモリ共有を可能にします。[b]
	内部共有可能	キャッシュ可。次のうちのいずれかです。[a] キャッシュ不可 ライトスルーキャッシュ可 ライトバック、書き込み割り当てキャッシュ可 ライトバック、書き込み割り当てなしキャッシュ可	複数のプロセッサ間で共有されるノーマルメモリを処理することを意図しています。
	共有不可	キャッシュ可。次のうちのいずれかです。[a] キャッシュ不可 ライトスルーキャッシュ可 ライトバック、書き込み割り当てキャッシュ可 ライトバック、書き込み割り当てなしキャッシュ可	単一プロセッサによってのみ使用されるノーマルメモリを処理することを意図しています。

a. キャッシュ可能属性は、内部および外部キャッシュ領域に対して独立して定義されます。
b. 外部共有可能属性の重要度は実装定義です。

A3.5.3 ARM アーキテクチャにおけるアトミック性

*アトミック性*は、*アトミック*アクセスと記述される、メモリアクセスの一機能です。ARM アーキテクチャの説明では、2 種類のアトミック性が使用されます。定義については、次を参照して下さい。

- *シングルコピーのアトミック性* P. A3-27
- *マルチコピーのアトミック性* P. A3-28

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996−1998, 2000, 2004−2008 ARM Limited. All rights reserved.

Single-copy atomicity

A read or write operation is *single-copy atomic* if the following conditions are both true:

- After any number of write operations to an operand, the value of the operand is the value written by one of the write operations. It is impossible for part of the value of the operand to come from one write operation and another part of the value to come from a different write operation.
- When a read operation and a write operation are made to the same operand, the value obtained by the read operation is one of:
  - — the value of the operand before the write operation
  - — the value of the operand after the write operation.

  It is never the case that the value of the read operation is partly the value of the operand before the write operation and partly the value of the operand after the write operation.

In ARMv7, the single-copy atomic processor accesses are:

- all byte accesses
- all halfword accesses to halfword-aligned locations
- all word accesses to word-aligned locations
- memory accesses caused by `LDREXD` and `STREXD` instructions to doubleword-aligned locations.

`LDM`, `LDC`, `LDC2`, `LDRD`, `STM`, `STC`, `STC2`, `STRD`, `PUSH`, `POP`, `RFE`, `SRS`, `VLDM`, `VLDR`, `VSTM`, and `VSTR` instructions are executed as a sequence of word-aligned word accesses. Each 32-bit word access is guaranteed to be single-copy atomic. A subsequence of two or more word accesses from the sequence might not exhibit single-copy atomicity.

Advanced SIMD element and structure loads and stores are executed as a sequence of accesses of the element or structure size. The element accesses are single-copy atomic if and only if both:

- the element size is 32 bits, or smaller
- the elements are naturally aligned.

Accesses to 64-bit elements or structures that are at least word-aligned are executed as a sequence of 32-bit accesses, each of which is single-copy atomic. Subsequences of two or more 32-bit accesses from the sequence might not be single-copy atomic.

When an access is not single-copy atomic, it is executed as a sequence of smaller accesses, each of which is single-copy atomic, at least at the byte level.

If an instruction is executed as a sequence of accesses according to these rules, some exceptions can be taken in the sequence and cause execution of the instruction to be abandoned. These exceptions are:

- synchronous Data Abort exceptions
- if low interrupt latency configuration is selected and the accesses are to Normal memory, see *Low interrupt latency configuration* on page B1-43:
  - — IRQ interrupts
  - — FIQ interrupts
  - — asynchronous aborts.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

シングルコピーのアトミック性

次の条件がいずれも TRUE の場合、読み出し / 書き込み操作は*シングルコピー アトミック*です。

- あるオペランドに対して任意の回数の書き込みの後、そのオペランドの値はその書き込み操作のうちのいずれか 1 つで書き込まれたものである。オペランドの値の一部はある書き込み操作によるもの、値の別の一部は別の書き込み操作によるものということは決して発生しない。
- 同じオペランドに対して読み出し操作と書き込み操作が実行される場合、読み出し操作によって得られる値が、次のいずれかである。
  - – 書き込み操作前のオペランドの値
  - – 書き込み操作後のオペランドの値

  読み出された値の一部は書き込み操作前のオペランドの値であり、別の一部は書き込み操作後のオペランドの値であるということは決して発生しない。

ARMv7 では、プロセッサによる次のアクセスがシングルコピー アトミックです。

- すべてのバイトアクセス
- ハーフワードアラインドの位置に対するすべてのハーフワードアクセス
- ワードアラインドの位置に対するすべてのワードアクセス
- ダブルワードアラインドの位置に対して `LDREXD` および `STREXD` 命令によって引き起こされるメモリアクセス

`LDM`、`LDC`、`LDC2`、`LDRD`、`STM`、`STC`、`STC2`、`STRD`、`PUSH`、`POP`、`RFE`、`SRS`、`VLDM`、`VLDR`、`VSTM`、`VSTR` 命令は、ワードアラインドのワードアクセスのシーケンスとして実行されます。各 32 ビット ワードアクセスは、シングルコピー アトミックであることが保証されます。このシーケンスに含まれる 2 つ以上のワードアクセスによるサブシーケンスは、シングルコピーのアトミック性を示さない可能性があります。

アドバンスト SIMD のエレメントと構造のロードおよびストアは、そのエレメントサイズまたは構造サイズのアクセスのシーケンスとして実行されます。エレメントアクセスは、次の条件の両方が満たされる場合、その場合のみ、シングルコピー アトミックです。

- エレメントサイズが 32 ビット以下である
- エレメントが自然にアラインしている

ワードアラインド以上の 64 ビットのエレメントまたは構造へのアクセスは、32 ビットアクセスのシーケンスとして実行され、それぞれがシングルコピー アトミックです。このシーケンスに含まれる 2 つ以上の 32 ビットアクセスのサブシーケンスは、シングルコピー アトミックでない場合があります。

アクセスがシングルコピー アトミックでない場合は、より小さいアクセスのシーケンスとして実行され、各シーケンスは少なくともバイトレベルではシングルコピーアトミックです。

命令がこれらの規則に従うアクセスのシーケンスとして実行される場合、シーケンス内でいくつかの例外処理を行うことが可能で、命令の実行が放棄されることがあります。これらの例外は次のとおりです。

- 同期データアボート例外

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If any of these exceptions are returned from using their preferred exception return, the instruction that generated the sequence of accesses is re-executed and so any accesses that had already been performed before the exception was taken are repeated.

Note

The exception behavior for these multiple access instructions means they are not suitable for use for writes to memory for the purpose of software synchronization.

For implicit accesses:

- Cache linefills and evictions have no effect on the single-copy atomicity of explicit transactions or instruction fetches.
- Instruction fetches are single-copy atomic for each instruction fetched.

  Note

  32-bit Thumb instructions are fetched as two 16-bit items.

- Translation table walks are performed as 32-bit accesses aligned to 32 bits, each of which is single-copy atomic.

Multi-copy atomicity

In a multiprocessing system, writes to a memory location are *multi-copy atomic* if the following conditions are both true:

- All writes to the same location are *serialized*, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a location does not return the value of a write until all observers observe that write.

Writes to Normal memory are not multi-copy atomic.

All writes to Device and Strongly-ordered memory that are single-copy atomic are also multi-copy atomic.

All write accesses to the same location are serialized. Write accesses to Normal memory can be repeated up to the point that another write to the same address is observed.

For Normal memory, serialization of writes does not prohibit the merging of writes.

A3.5.4 Normal memory

Normal memory is idempotent, meaning that it exhibits the following properties:

- read accesses can be repeated with no side effects
- repeated read accesses return the last value written to the resource being read
- read accesses can prefetch additional memory locations with no side effects

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- 低割り込みレイテンシ構成が選択されていてノーマルメモリに対するアクセスの場合（*低割り込みレイテンシ構成* P. B1-43 を参照）
  - IRQ 割り込み
  - FIQ 割り込み
  - 非同期アボート

これらの例外のいずれかが、推奨の例外からの復帰方法により復帰した場合は、アクセスのシーケンスを生成した命令が再実行されるため、例外の取得前にすでに実行されたアクセスが繰り返し実行されます。

注

これらの複数アクセス命令に対する例外動作のため、これらの命令はソフトウェア同期化を目的としたメモリへの書き込みには適していません。

暗黙アクセスについては、次の規則が適用されます。

- キャッシュのラインフィルおよび退出は、明示的なトランザクションや命令フェッチのシングルコピーのアトミック性に影響を与えません。
- 命令フェッチは、フェッチされた各命令に対してシングルコピー アトミックです。

  注

  32 ビット Thumb 命令は、2 つの 16 ビット項目としてフェッチされます。

- 変換テーブルウォークは、32 ビットにアラインした 32 ビットアクセスとして実行され、それぞれがシングルコピー アトミックです。

マルチコピーのアトミック性

マルチプロセッシングシステムでは、次の条件がいずれも TRUE の場合、メモリ位置への書き込みは *マルチコピー アトミック*です。

- 同じ位置への書き込みはすべて*直列化*される。これは、すべてのオブザーバから見て、書き込みが同じ順序で観測されることを意味します。ただし、一部のオブザーバがすべての書き込みを観測しない可能性はあります。
- すべてのオブザーバによって書き込みが観測されるまで、ある位置からの読み出しがその書き込みの値を返すことはない。

ノーマルメモリへの書き込みはマルチコピー アトミックではありません。

デバイスおよびストロングリオーダ メモリへのシングルコピー アトミックであるすべての書き込みは、マルチコピー アトミックでもあります。

同じ位置への書き込みアクセスはすべて直列化されます。ノーマルメモリへの書き込みアクセスは、同じアドレスへの別の書き込みが観測されるポイントまで繰り返すことが可能です。

ノーマルメモリの場合は、書き込みの直列化により書き込みのマージが禁止されることはありません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- write accesses can be repeated with no side effects, provided that the contents of the location are unchanged between the repeated writes
- unaligned accesses can be supported
- accesses can be merged before accessing the target memory system.

Normal memory can be read/write or read-only, and a Normal memory region is defined as being either Shareable or Non-shareable. In a VMSA implementation, Shareable Normal memory can be either Inner Shareable or Outer Shareable. In a PMSA implementation, no distinction is made between Inner Shareable and Outer Shareable regions.

The Normal memory type attribute applies to most memory used in a system.

Accesses to Normal Memory have a weakly consistent model of memory ordering. See a standard text describing memory ordering issues for a description of weakly consistent memory models, for example chapter 2 of *Memory Consistency Models for Shared Memory-Multiprocessors*, Kourosh Gharachorloo, Stanford University Technical Report CSL-TR-95-685. In general, for Normal memory, barrier operations are required where the order of memory accesses observed by other observers must be controlled. This requirement applies regardless of the cacheablility and shareability attributes of the Normal memory region.

The ordering requirements of accesses described in *Ordering requirements for memory accesses* on page A3-45 apply to all explicit accesses.

An instruction that generates a sequence of accesses as described in *Atomicity in the ARM architecture* on page A3-26 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

The architecture permits speculative accesses to memory locations marked as Normal if the access permissions and domain permit an access to the locations.

A Normal memory region has shareability attributes that define the data coherency properties of the region. These attributes do not affect the coherency requirements of:

- instruction fetches, see *Instruction coherency issues* on page A3-53
- translation table walks, if supported, in the base ARMv7 architecture and in versions of the architecture before ARMv7, see *TLB maintenance operations and the memory order model* on page B3-59.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.5.4 ノーマルメモリ

ノーマルメモリは「ベキ等」です。つまり、次の属性を示します。

- 読み出しアクセスは副作用なしに繰り返し可能である。
- 繰り返された読み出しアクセスでは、読み出されたリソースに最後に書き込まれた値が返される。
- 読み出しアクセスは、副作用なしに追加のメモリ位置をプリフェッチ可能である。
- 書き込みアクセスは、書き込みを繰り返す間にその位置の内容が変化しない限り、副作用なしに繰り返し可能である。
- アンアラインドアクセスをサポート可能である。
- ターゲット メモリシステムへのアクセス前にアクセスをマージ可能である。

ノーマルメモリは、読み出し / 書き込みまたは読み出し専用であり、ノーマルメモリ領域は共有可能または共有不可のいずれかとして定義されます。VMSA 実装において、共有可能ノーマルメモリは内部共有可能か外部共有可能のいずれかです。PMSA 実装において、内部共有可能領域と外部共有可能領域は区別されません。

ノーマルメモリのタイプ属性は、システムで使用される大部分のメモリに適用されます。

ノーマルメモリへのアクセスは、メモリオーダリングの弱い整合性モデルを持ちます。弱い整合性を持つメモリモデルの説明については、メモリオーダリング問題を説明している標準的なテキストを参照して下さい。例えば、『*Memory Consistency Models for Shared Memory-Multiprocessors*』(Kourosh Gharachorloo, Stanford University Technical Report CSL-TR-95-685) の第 2 章を参照して下さい。一般的に、通常メモリの場合は、他のオブザーバによって観測されるメモリアクセスの順序を制御する必要がある場所で、バリア操作が必要になります。この要件は、ノーマルメモリ領域のキャッシュ可能属性および共有可能属性にかかわらず適用されます。

メモリアクセスの順序付けの要件 P. A3-45 で説明されているアクセスのオーダリング要件は、すべての明示的なアクセスに適用されます。

ARM アーキテクチャにおけるアトミック性 P. A3-26 で説明されているアクセスのシーケンスを生成する命令は、アクセスのシーケンス中に取得された例外の結果として破棄されることがあります。例外からの復帰時に命令が再始動され、1 つ以上のメモリ位置が複数回アクセスされることがあります。この場合、書き込みアクセス間で変更された位置への書き込みアクセスが繰り返される可能性があります。

アーキテクチャでは、ノーマルとしてマークされたメモリ位置への投機的アクセスが許可されています。ただし、アクセス許可とドメインによりその位置へのアクセスが許可されている場合に限られます。

ノーマルメモリ領域には、領域のデータコヒーレンシ特性を定義する共有可能属性があります。これらの属性は、次のコヒーレンシ要件に影響を与えません。

- 命令フェッチ（*命令コヒーレンシの問題* P. A3-53 参照）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Non-shareable Normal memory

For a Normal memory region, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single processor.

A region of Normal memory with the Non-shareable attribute does not have any requirement to make data accesses by different observers coherent, unless the memory is non-cacheable. If other observers share the memory system, software must use cache maintenance operations if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives do not take account of the possibility of accesses by more than one observer.

Shareable, Inner Shareable, and Outer Shareable Normal memory

For Normal memory, the Shareable and Outer Shareable memory attributes describe Normal memory that is expected to be accessed by multiple processors or other system masters:

- In a VMSA implementation, Normal memory that has the Shareable attribute but not the Outer Shareable attribute assigned is described as having the Inner Shareable attribute.
- In a PMSA implementation, no distinction is made between Inner Shareable and Outer Shareable Normal memory, and you cannot assign the Outer Shareable attribute to Normal memory regions.

A region of Normal memory with the Shareable attribute is one for which data accesses to memory by different observers within the same shareability domain are coherent.

The Outer Shareable attribute is introduced in ARMv7, and can be applied only to a Normal memory region in a VMSA implementation that has the Shareable attribute assigned. It creates three levels of shareability for a Normal memory region:

Non-shareable

A Normal memory region that does not have the Shareable attribute assigned.

Inner Shareable

A Normal memory region that has the Shareable attribute assigned, but not the Outer Shareable attribute.

Outer Shareable

A Normal memory region that has both the Shareable and the Outer Shareable attributes assigned.

These attributes can be used to define sets of observers for which the shareability attributes make the data or unified caches transparent for data accesses. The sets of observers that are affected by the shareability attributes are described as *shareability domains*. The details of the use of these attributes are system-specific. Example A3-1 on page A3-31 shows how they might be used:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- サポートされている場合に限り、ベースとなる ARMv7 アーキテクチャおよび ARMv7 以前のアーキテクチャバージョンの変換テーブルウォーク（*TLB 保守操作とメモリオーダモデル* P. B3- 59 参照）

共有不可のノーマルメモリ

ノーマルメモリ領域に対して、共有不可属性は、そのノーマルメモリにアクセスするプロセッサが 1 つだけであると予測されることを示しています。

共有不可属性を持つノーマルメモリの領域には、メモリがキャッシュ不可でない限り、異なるオブザーバによるデータアクセスをコヒーレントにしなければならないという要件はありません。他のオブザーバによってメモリシステムが共有されており、キャッシュの存在がオブザーバ間の通信の際にコヒーレンシの問題を引き起こす可能性がある場合は、ソフトウェアでキャッシュ保守操作を使用する必要があります。このキャッシュ保守要件は、メモリオーダリングを保証するために必要なバリア操作に追加されます。

共有不可のノーマルメモリに対して、排他ロード / 排他ストアの同期化基本命令では、複数のオブザーバによるアクセスの可能性は考慮されません。

共有可能、内部共有可能、外部共有可能のノーマルメモリ

ノーマルメモリの場合、共有可能および外部共有可能のメモリ属性は、複数のプロセッサまたは他のシステムマスタによってアクセスされる可能性のあるノーマルメモリを示します。

- VMSA 実装において、共有可能属性を持ち、外部共有可能属性を持たないノーマルメモリは、内部共有可能属性を持つと記述されます。
- PMSA 実装において、内部共有可能ノーマルメモリと外部共有可能ノーマルメモリは区別されず、ノーマルメモリ領域に外部共有可能属性を割り当てることはできません。

共有可能属性を持つノーマルメモリ領域とは、同じ共有可能ドメイン内の異なるオブザーバによるメモリへのデータアクセスがコヒーレントである領域です。

外部共有可能属性は ARMv7 で導入されました。この属性は、VMSA 実装に含まれ、共有可能属性が割り当てられているノーマルメモリ領域にのみ適用可能です。これにより、ノーマルメモリ領域には 3 つのレベルの共有可能性があることになります。

共有不可

共有可能属性が割り当てられていないノーマルメモリ領域

内部共有可能

共有可能属性が割り当てられているが、外部共有可能属性は割り当てられていないノーマルメモリ領域

外部共有可能

共有可能属性と外部共有可能属性の両方が割り当てられているノーマルメモリ領域

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Example A3-1 Use of shareability attributes

In a VMSA implementation, a particular sub-system with two clusters of processors has the requirement that:

- in each cluster, the data or unified caches of the processors in the cluster are transparent for all data accesses with the Inner Shareable attribute
- however, between the two clusters, the caches:
  - — are not transparent for data accesses that have only the Inner Shareable attribute
  - — are transparent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different shareability domain for the Inner Shareable attribute, but all components of the sub-system are in the same shareability domain for the Outer Shareable attribute.

A system might implement two such sub-systems. If the data or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable shareability domains.

Having two levels of shareability attribute means you can reduce the performance and power overhead for shared memory regions that do not need to be part of the Outer Shareable shareability domain.

Whether an ARMv7 implementation supports the Outer Shareable attribute is IMPLEMENTATION DEFINED. If the Outer Shareable attribute is supported, its significance in the implementation is IMPLEMENTATION DEFINED.

For Shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

Note

The Shareable concept enables system designers to specify the locations in Normal memory that must have coherency requirements. However, to facilitate porting of software, software developers must not assume that specifying a memory region as Non-shareable permits software to make assumptions about the incoherency of memory locations between different processors in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that make use of the Shareable concept. Any multiprocessing implementation might implement caches that, inherently, are shared between different processing elements.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

これらの属性を使用して定義したオブザーバのセットでは、共有可能属性によってデータキャッシュまたは統一キャッシュがデータアクセスにおいて透過的になります。このように共有可能属性に影響を受けるオブザーバのセットは、*共有可能ドメイン*と呼ばれます。これらの属性の詳細な用法は、各システムに固有です。可能な使用法を、例 A3-1 P. A3-31 に示します。

例 A3-1 共有可能属性の使用法

VMSA 実装では、2 つのクラスタのプロセッサを持つ特定のサブシステムについて次の要件があります。

- 各クラスタにおいて、クラスタ内のプロセッサのデータキャッシュまたは統一キャッシュは、内部共有可能属性を持つすべてのデータアクセスについて透過的であること
- ただし、2 つのクラスタの間においてキャッシュは次の条件を満たすこと
  - – 内部共有可能属性のみを持つデータアクセスについては透過的にならない
  - – 外部共有可能属性を持つデータアクセスについては透過的になる

このシステムで、各クラスタは内部共有可能属性の別の共有可能ドメインにありますが、サブシステムのすべてのコンポーネントは外部共有可能属性の同じ共有可能ドメインにあります。

システムは、このようなサブシステムを 2 つ実装する可能性があります。1 つのサブシステムのデータキャッシュまたは統一キャッシュが別のサブシステムからのアクセスに透過的でない場合、このシステムは 2 つの外部共有可能ドメインを持っています。

2 つのレベルの共有可能属性を持つということは、外部共有可能ドメインの一部となる必要のない共有メモリ領域について、性能と消費電力のオーバヘッドを減らせることを意味します。

ARMv7 の実装が外部共有可能属性をサポートするかどうかは実装定義です。外部共有可能属性がサポートされている場合、実装における重要度は実装定義です。

共有可能なノーマルメモリに対して、排他ロード / 排他ストアの同期化基本命令では、同じ共有可能ドメインに対して複数のオブザーバによるアクセスの可能性が考慮されます。

注

共有可能という概念を使用すると、システム設計者はノーマルメモリ内でコヒーレンシ要件を持つ必要のある位置を指定できます。ただし、ソフトウェアの移植を容易にするため、ソフトウェア開発者は、あるメモリ領域を共有不可と指定しても、共有メモリシステムでの異なるプロセッサ間でメモリ位置にコヒーレンシがないという前提をソフトウェアで行うことが許可されていると想定すべきではありません。このような想定は、共有可能の概念を使用している異なるマルチプロセッシング実装間では移植不能です。どのマルチプロセッシング実装も、異なるプロセッシングエレメント間で本質的に共有されるキャッシュが実装されている可能性があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Write-Through Cacheable, Write-Back Cacheable and Non-cacheable Normal memory

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory can be marked as being one of:

- Write-Through Cacheable
- Write-Back Cacheable, with an additional qualifier that marks it as one of:
  - — Write-Back, Write-Allocate
  - — Write-Back, no Write-Allocate
- Non-cacheable.

If the same memory locations are marked as having different cacheability attributes, for example by the use of aliases in a virtual to physical address mapping, behavior is UNPREDICTABLE.

The cacheability attributes provide a mechanism of coherency control with observers that lie outside the shareability domain of a region of memory. In some cases, the use of Write-Through Cacheable or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. To this end, the architecture requires the following properties for Non-cacheable or Write-Through Cacheable memory:

- a completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance

- a completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.

Note

Implementations can also use the cacheability attributes to provide a performance hint regarding the performance benefit of caching. For example, it might be known to a programmer that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between Write-Back Write-Allocate and Write-Back no Write-Allocate memory exists only as a hint for performance.

The ARM architecture provides independent cacheability attributes for Normal memory for two conceptual levels of cache, the *inner* and the *outer* cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

- inner refers to the innermost caches, and always includes the lowest level of cache

- no cache controlled by the Inner cacheability attributes can lie outside a cache controlled by the Outer cacheability attributes

- an implementation might not have any outer cache.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ライトスルーキャッシュ可、ライトバックキャッシュ可、およびキャッシュ不可のノーマルメモリ

ノーマルメモリの各領域は、外部共有可能、内部共有可能、共有不可に加え、次のうちのいずれかとしてマークされることがあります。

- ライトスルーキャッシュ可
- ライトバックキャッシュ可。この場合、追加の修飾子によってさらに次のいずれかにマークされます。
  - ー ライトバック、書き込み割り当て
  - ー ライトバック、書き込み割り当てなし
- キャッシュ不可

同じメモリ位置が、仮想アドレスから物理アドレスへのマッピングにおけるエイリアスの使用などによって、異なるキャッシュ可能属性を持つとマークされている場合、動作は予測不能です。

キャッシュ可能属性は、コヒーレンシ制御の機構を提供します。オブザーバはメモリ領域の共有可能ドメインの外に置きます。メモリのライトスルーキャッシュ可能領域またはキャッシュ不可領域を使用したほうが、ハードウェアのコヒーレンシ機構やキャッシュ保守ルーチンを使用するよりも、コヒーレンシを制御する有効な機構となる場合もあります。この目的で、アーキテクチャはキャッシュ不可またはライトスルーキャッシュ可能メモリに次の特性を要求しています。

- 1 つのレベルのキャッシュについてキャッシュ不可またはライトスルーキャッシュ可能であるメモリ位置に対して、そのキャッシュのレベル内のメモリシステムにアクセスするオブザーバが書き込みを完了すると、それは、明示的なキャッシュ保守を実行しなくても、そのキャッシュのレベルの外にあるメモリシステムにアクセスするすべてのオブザーバに認識されます。

- 1 つのレベルのキャッシュについてキャッシュ不可であるメモリ位置に対して、そのキャッシュのレベル外のメモリシステムにアクセスするオブザーバが書き込みを完了すると、それは、明示的なキャッシュ保守を実行しなくても、そのキャッシュのレベル内にあるメモリシステムにアクセスするすべてのオブザーバに認識されます。

注

実装でキャッシュ可能属性を使い、キャッシュの性能上の利点に関するパフォーマンスヒントを提供することもできます。これによって例えばプログラマは、メモリのある部分が二度とアクセスされず、キャッシュ不可として扱ったほうが良いことがわかります。ライトバック書き込み割り当てメモリとライトバック書き込み割り当てなしメモリの区別は、パフォーマンスを考慮するヒントとしてのみ存在します。

ARM アーキテクチャには、*内部*キャッシュ、*外部*キャッシュという 2 つの概念レベルに関して、ノーマルメモリに独立したキャッシュ可能属性があります。これらのキャッシュの概念レベルと、実装されているキャッシュの物理レベルとの関係は実装定義で、内部共有可能ドメインと外部共有可能ドメインの境界とは異なる場合があります。ただし、次のような制限があります。

- 内部とは、最も内側のキャッシュを指し、常に最低レベルのキャッシュを含みます。

- 内部キャッシュ可能属性によって制御されるキャッシュは、外部キャッシュ可能属性によって制御されるキャッシュの外に存在することはできません。

- 実装には外部キャッシュが存在しないこともあります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Example A3-2 to Example A3-4 describe the three possible ways of implementing a system with three levels of cache, L1 to L3. L1 is the level closest to the processor, see *Memory hierarchy* on page A3-52.

Example A3-2 Implementation with two inner and one outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:
- the Inner cacheability attribute applied to L1 and L2 cache
- the Outer cacheability attribute applied to L3 cache.

Example A3-3 Implementation with three inner and no outer cache levels

Implement the three levels of cache in the system, L1 to L3, with the Inner cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer cacheability attribute.

Example A3-4 Implementation with one inner and two outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:
- the Inner cacheability attribute applied to L1 cache
- the Outer cacheability attribute applied to L2 and L3 cache.

A3.5.5 Device memory

The Device memory type attribute defines memory locations where an access to the location can cause side effects, or where the value returned for a load can vary depending on the number of loads performed. Memory-mapped peripherals and I/O locations are examples of memory regions normally marked as being Device memory.

For explicit accesses from the processor to memory marked as Device:
- all accesses occur at their program size
- the number of accesses is the number specified by the program.

An implementation must not repeat an access to a Device memory location if the program has only one access to that location. In other words, accesses to Device memory locations are not restartable.

The architecture does not permit speculative accesses to memory marked as Device.

The architecture permits an Advanced SIMD element or structure load instruction to access bytes in Device memory that are not explicitly accessed by the instruction, provided the bytes accessed are within a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

Address locations marked as Device are never held in a cache.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

例 A3-2 から例 A3-4 までは、L1 から L3 までの 3 レベルのキャッシュを持つシステムを実装する 3 つの方法を示しています。L1 が最もプロセッサに近いキャッシュです。*メモリの階層* P. A3-52 を参照して下さい。

例 A3-2 2 つの内部キャッシュレベルと 1 つの外部キャッシュレベルを持つ実装

システムに L1 から L3 までの 3 レベルのキャッシュを実装
- 内部キャッシュ可能属性を L1 キャッシュと L2 キャッシュに適用
- 外部キャッシュ可能属性を L3 キャッシュに適用

例 A3-3 3 つの内部キャッシュレベルを持ち、外部キャッシュレベルのない実装

システムに L1 から L3 までの 3 レベルのキャッシュを実装し、内部キャッシュ可能属性を L1、L2、L3 キャッシュに適用します。外部キャッシュ可能属性は使用しません。

例 A3-4 1 つの内部キャッシュレベルと 2 つの外部キャッシュレベルを持つ実装

システムに L1 から L3 までの 3 レベルのキャッシュを実装
- 内部キャッシュ可能属性を L1 キャッシュに適用
- 外部キャッシュ可能属性を L2 および L3 キャッシュに適用

A3.5.5 デバイスメモリ

デバイスメモリのタイプ属性は、メモリ位置へのアクセスが副作用を引き起こす、またはロードに対して返される値が実行されたロードの回数に応じて変化するようなメモリ位置を示します。メモリマップされたペリフェラルおよび I/O 位置は、一般にデバイスメモリとしてマークされるメモリ領域の例です。

デバイスとしてマークされているメモリに対してプロセッサから明示的なアクセスを行う場合、次の規則が適用されます。
- すべてのアクセスはプログラムで指定されたサイズで発生します。
- アクセスの回数は、プログラムによって指定された回数と等しくなります。

プログラムが一回しかアクセスしていなければ、実装は対応するデバイスメモリ位置へのアクセスを繰り返してはいけません。言い換えれば、デバイスメモリ位置へのアクセスは再始動可能ではありません。

アーキテクチャでは、デバイスとしてマークされたメモリへの投機的アクセスは許可されていません。

アーキテクチャでは、アドバンスト SIMD のエレメントまたは構造ロード命令が、命令によって明示的にアクセスされないデバイスメモリ内のバイトにアクセスすることが許可されています。ただし、アクセスされるバイトは、16 バイトのウィンドウ内で、16 バイトでアラインし、命令によって明示的にアクセスされるバイトを 1 つ以上含む必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

All explicit accesses to Device memory must comply with the ordering requirements of accesses described in *Ordering requirements for memory accesses* on page A3-45.

An instruction that generates a sequence of accesses as described in *Atomicity in the ARM architecture* on page A3-26 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

Note

Do not use an instruction that generates a sequence of accesses to access Device memory if the instruction might generate an abort on any access other than the first one.

Any unaligned access that is not faulted by the alignment restrictions and accesses Device memory has UNPREDICTABLE behavior.

Shareable attribute for Device memory regions

Device memory regions can be given the Shareable attribute. This means that a region of Device memory can be described as either:

- Shareable Device memory
- Non-shareable Device memory.

Non-shareable Device memory is defined as only accessible by a single processor. An example of a system supporting Shareable and Non-shareable Device memory is an implementation that supports both:

- a local bus for its private peripherals
- system peripherals implemented on the main shared system bus.

Such a system might have more predictable access times for local peripherals such as watchdog timers or interrupt controllers. In particular, a specific address in a Non-shareable Device memory region might access a different physical peripheral for each processor.

A3.5.6 Strongly-ordered memory

The Strongly-ordered memory type attribute defines memory locations where an access to the location can cause side effects, or where the value returned for a load can vary depending on the number of loads performed. Examples of memory regions normally marked as being Strongly-ordered are memory-mapped peripherals and I/O locations.

For explicit accesses from the processor to memory marked as Strongly-ordered:

- all accesses occur at their program size
- the number of accesses is the number specified by the program.

An implementation must not repeat an access to a Strongly-ordered memory location if the program has only one access to that location. In other words, accesses to Strongly-ordered memory locations are not restartable.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

デバイスとしてマークされているアドレス位置はキャッシュに保持されません。

デバイスメモリへのすべての明示的アクセスは、*メモリアクセスの順序付けの要件* P. A3-45 で説明されているアクセスの順序付け要件に従う必要があります。

ARM アーキテクチャにおけるアトミック性 P. A3-26 で説明されているアクセスのシーケンスを生成する命令は、アクセスのシーケンス中に取得された例外の結果として破棄されることがあります。例外からの復帰時に命令が再始動され、1 つ以上のメモリ位置が複数回アクセスされることがあります。この場合、書き込みアクセス間で変更された位置への書き込みアクセスが繰り返される可能性があります。

注

命令が最初以外のアクセスでアボートを生成する可能性がある場合は、デバイスメモリにアクセスするアクセスのシーケンスを生成する命令は使用しないで下さい。

アライメント制約事項によってフォールトとされていないアンアラインドアクセスがデバイスメモリに対して行われた場合、動作は予測不能です。

デバイスメモリ領域に対する共有能可属性

デバイスメモリ領域には、共有可能属性を与えることができます。これは、デバイスメモリの領域に次のいずれかを指定できることを意味します。

- 共有可能のデバイスメモリ
- 共有不可のデバイスメモリ

共有不可のデバイスメモリは、単一プロセッサによってのみアクセス可能なものとして定義されます。共有可能および共有不可のデバイスメモリをサポートするシステムの例として、次の両方をサポートする実装が挙げられます。

- 専用ペリフェラル用のローカルバス
- 共有されるメインのシステムバス上に実装されたシステムペリフェラル

このようなシステムでは、ウォッチドッグタイマまたは割り込みコントローラなどのローカルペリフェラルに対するアクセスタイムが予測しやすい可能性があります。特に、共有不可デバイスメモリ領域の特定のアドレスは、プロセッサごとに別の物理ペリフェラルにアクセスする可能性があります。

A3.5.6 ストロングリオーダ メモリ

ストロングリオーダ メモリのタイプ属性は、メモリ位置へのアクセスが副作用を引き起こす、またはロードに対して返される値が実行されたロードの回数に応じて変化する可能性があるようなメモリ位置を示します。一般にストロングリオーダとマークされるメモリ領域の例として、メモリマップされたペリフェラルや I/O 位置があります。

プロセッサから、ストロングリオーダとしてマークされているメモリへ明示的なアクセスを行う場合、次の規則が適用されます。

- すべてのアクセスはプログラムで指定されたサイズで発生します。
- アクセスの回数は、プログラムによって指定された回数と等しくなります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The architecture does not permit speculative accesses to memory marked as Strongly-ordered.

The architecture permits an Advanced SIMD element or structure load instruction to access bytes in Strongly-ordered memory that are not explicitly accessed by the instruction, provided the bytes accessed are within a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

Address locations in Strongly-ordered memory are not held in a cache, and are always treated as Shareable memory locations.

All explicit accesses to Strongly-ordered memory must correspond to the ordering requirements of accesses described in *Ordering requirements for memory accesses* on page A3-45.

An instruction that generates a sequence of accesses as described in *Atomicity in the ARM architecture* on page A3-26 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

Note

Do not use an instruction that generates a sequence of accesses to access Strongly-ordered memory if the instruction might generate an abort on any access other than the first one.

Any unaligned access that is not faulted by the alignment restrictions and accesses Strongly-ordered memory has UNPREDICTABLE behavior.

Note

See *Ordering of instructions that change the CPSR interrupt masks* on page AppxG-8 for additional requirements that apply to accesses to Strongly-ordered memory in ARMv6.

A3.5.7 Memory access restrictions

The following restrictions apply to memory accesses:

- For any access X, the bytes accessed by X must all have the same memory type attribute, otherwise the behavior of the access is UNPREDICTABLE. That is, an unaligned access that spans a boundary between different memory types is UNPREDICTABLE.
- For any two memory accesses X and Y that are generated by the same instruction, the bytes accessed by X and Y must all have the same memory type attribute, otherwise the results are UNPREDICTABLE. For example, an `LDC`, `LDM`, `LDRD`, `STC`, `STM`, or `STRD` that spans a boundary between Normal and Device memory is UNPREDICTABLE.
- An instruction that generates an unaligned memory access to Device or Strongly-ordered memory is UNPREDICTABLE.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

実装では、プログラムがストロングリオーダのメモリ位置へ 1 回しかアクセスしない場合、その位置へのアクセスを繰り返してはいけません。言い換えれば、ストロングリオーダメモリ位置へのアクセスは再始動可能ではありません。

アーキテクチャでは、ストロングリオーダとしてマークされたメモリへの投機的アクセスは許可されていません。

アーキテクチャでは、アドバンスト SIMD のエレメントまたは構造ロード命令が、命令によって明示的にアクセスされないストロングリオーダメモリ内のバイトにアクセスすることが許可されています。ただし、アクセスされるバイトは、16 バイトのウィンドウ内で、16 バイトでアラインし、命令によって明示的にアクセスされるバイトを 1 つ以上含む必要があります。

ストロングリオーダメモリのアドレス位置はキャッシュに保持されず、常に共有可能メモリ位置として扱われます。

ストロングリオーダ メモリへのすべての明示的アクセスは、*メモリアクセスの順序付けの要件* P. A3-45 で説明されているアクセスのオーダリング要件に適合している必要があります。

ARM アーキテクチャにおけるアトミック性 P. A3-26 で説明されているアクセスのシーケンスを生成する命令は、アクセスのシーケンス中に取得された例外の結果として破棄されることがあります。例外からの復帰時に命令が再始動され、1 つ以上のメモリ位置が複数回アクセスされることがあります。この場合、書き込みアクセス間で変更された位置への書き込みアクセスが繰り返される可能性があります。

注

命令が最初以外のアクセスでアボートを生成する可能性がある場合は、ストロングリオーダ メモリにアクセスするアクセスのシーケンスを生成する命令は使用しないで下さい。

アライメント制約事項によってフォールトとされていないアンアラインドアクセスがストロングリオーダ メモリに対して行われた場合、動作は予測不能です。

注

ARMv6 においてストロングリオーダ メモリへのアクセスに適用される他の要件については、*CPSR 割り込みマスクを変更する命令の順序付け* P. 付録 G-8 を参照して下さい。

A3.5.7 メモリアクセスの制約事項

メモリアクセスには、次の制約事項が適用されます。

- 任意のアクセス X について、X によってアクセスされるバイトはすべて同じメモリタイプ属性を持つ必要があり、そうでない場合アクセスの動作は予測不能です。つまり、異なるメモリタイプ間の境界をまたぐアンアラインドアクセスは予測不能です。
- 同じ命令によって生成された 2 つのメモリアクセス X および Y について、X および Y によってアクセスされるバイトは同じメモリタイプ属性を持つ必要があり、そうでない場合結果は予測不能です。例えば、通常メモリとデバイスメモリとの間をまたぐ `LDC`、`LDM`、`LDRD`、`STC`、`STM`、`STRD` 命令は予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- To ensure access rules are maintained, an instruction that causes multiple accesses to Device or Strongly-ordered memory must not cross a 4KB address boundary, otherwise the effect is UNPREDICTABLE. For this reason, it is important that an access to a volatile memory device is not made using a single instruction that crosses a 4KB address boundary.

  ARM expects this restriction to impose constraints on the placing of volatile memory devices in the memory map of a system, rather than expecting a compiler to be aware of the alignment of memory accesses.

- For instructions that generate accesses to Device or Strongly-ordered memory, implementations must not change the sequence of accesses specified by the pseudocode of the instruction. This includes not changing:
  — how many accesses there are
  — the time order of the accesses
  — the data sizes and other properties of each access.

  In addition, processor implementations expect any attached memory system to be able to identify the memory type of an accesses, and to obey similar restrictions with regard to the number, time order, data sizes and other properties of the accesses.

  Exceptions to this rule are:

  — An implementation of a processor can break this rule, provided that the information it supplies to the memory system enables the original number, time order, and other details of the accesses to be reconstructed. In addition, the implementation must place a requirement on attached memory systems to do this reconstruction when the accesses are to Device or Strongly-ordered memory.

    For example, an implementation with a 64-bit bus might pair the word loads generated by an LDM into 64-bit accesses. This is because the instruction semantics ensure that the 64-bit access is always a word load from the lower address followed by a word load from the higher address. However the implementation must permit the memory systems to unpack the two word loads when the access is to Device or Strongly-ordered memory.

  — Any implementation technique that produces results that cannot be observed to be different from those described above is legitimate.

  — An Advanced SIMD element or structure load instruction can access bytes in Device or Strongly-ordered memory that are not explicitly accessed by the instruction, provided the bytes accessed are within a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

- Any multi-access instruction that loads or stores the PC must access only Normal memory. If the instruction accesses Device or Strongly-ordered memory the result is UNPREDICTABLE. There is one exception to this restriction. In the VMSA architecture, when the MMU is disabled any multi-access instruction that loads or stores the PC functions correctly, see *Enabling and disabling the MMU* on page B3-5.

- Any instruction fetch must access only Normal memory. If it accesses Device or Strongly-ordered memory, the result is UNPREDICTABLE. For example, instruction fetches must not be performed to an area of memory that contains read-sensitive devices, because there is no ordering requirement between instruction fetches and explicit accesses.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- デバイスメモリまたはストロングリオーダ メモリへのアンアラインド メモリアクセスを生成する命令は、予測不能です。

- アクセス規則を維持するため、デバイスメモリまたはストロングリオーダ メモリへの複数のアクセスを引き起こす命令は、4KB のアドレス境界をまたぐことはできません。またいだ場合の影響は予測不能です。このため重要な点として、4KB のアドレス境界にまたがる単一の命令を使用して揮発性メモリデバイスにアクセスすることはできません。

  この制約は、システムのメモリマップにそのようなデバイスを配置する際の制限を意図したもので、コンパイラがメモリアクセスのアライメントを認識することを前提とはしていません。

- デバイスメモリまたはストロングリオーダ メモリへのアクセスを生成する命令の場合、命令の擬似コードによって指定されるアクセスのシーケンスを実装で変更することはできません。この条件には、次の内容を変更しないことが含まれます。
  – アクセスの回数
  – アクセスの時間的な順序
  – 各アクセスのデータサイズおよび他のプロパティ

  さらに、プロセッサの実装では、接続されている任意のメモリシステムがアクセスのメモリタイプを識別し、アクセスの回数、時間的な順序、データサイズ、および他のプロパティに関する同様の制約事項に従うことが想定されます。

  この規則の例外は、次のとおりです。

  – あるプロセッサの実装で、メモリシステムに提供される情報を使って、アクセスの元の回数、時間的な順序、および他の詳細を再構築することが可能であれば、この規則を守る必要はありません。さらに、その実装ではアクセスがデバイスメモリまたはストロングリオーダ メモリに対して行われる場合、接続されているメモリシステムがこの再構築を実行することを要件とする必要があります。

    例えば、64 ビットバスを使用する実装では、LDM 命令によって生成されたワードロードを 64 ビットアクセスにペアリングします。これは、命令セマンティクスにより、64 ビットアクセスでは常に、下位アドレスからのワードロードに続いて上位アドレスからのワードロードの順序で実行されることが保証されているためです。ただし、実装ではデバイスメモリまたはストロングリオーダ メモリへのアクセスが実行される場合に、メモリシステムが 2 つのワードロードをアンパックすることが許可される必要があります。

  – どのような場合でも、上で説明されたものと異なる結果が観測されなければ、任意の実装は正当なものです。

  – アドバンスト SIMD のエレメントまたは構造ロード命令は、命令によって明示的にアクセスされないデバイスメモリまたはストロングリオーダ メモリ内のバイトにアクセスすることができます。ただし、アクセスされるバイトは、16 バイトのウィンドウ内で、16 バイトでアラインし、命令によって明示的にアクセスされるバイトを 1 つ以上含む必要があります。

- PC をロード / ストアするマルチアクセス命令では、ノーマルメモリのみにアクセスできます。デバイスメモリまたはストロングリオーダ メモリへのアクセス命令が実行された場合、結果は予測不能です。この制約事項には 1 つ例外があります。VMSA アーキテクチャでは、MMU が非稼動の場合、PC をロードまたはストアするマルチアクセス命令はすべて正常に機能します。*MMU の許可と禁止* P. B3-5 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- Behavior is UNPREDICTABLE if the same memory location:
  - — is marked as Shareable Normal and Non-shareable Normal
  - — is marked as having different memory types (Normal, Device, or Strongly-ordered)
  - — is marked as having different cacheability attributes
  - — is marked as being Shareable Device and Non-shareable Device memory.

  Such memory marking contradictions can occur, for example, by the use of aliases in a virtual to physical address mapping.

Before ARMv6, it is IMPLEMENTATION DEFINED whether a low interrupt latency mode is supported. From ARMv6, low interrupt latency support is controlled by the SCTLR.FI bit. It is IMPLEMENTATION DEFINED whether multi-access instructions behave correctly in low interrupt latency configurations.

A3.5.8 Backwards compatibility

From ARMv6, the memory attributes are significantly different from those in previous versions of the architecture. Table A3-9 shows the interpretation of the earlier memory types in the light of this definition.

Table A3-9 Backwards compatibility

Previous architectures	ARMv6 and ARMv7 attribute
NCNB (Non-cacheable, Non-bufferable)	Strongly-ordered
NCB (Non-cacheable, Bufferable)	Shareable Device
Write-Through Cacheable, Bufferable	Non-shareable Normal, Write-Through Cacheable
Write-Back Cacheable, Bufferable	Non-shareable Normal, Write-Back Cacheable

A3.5.9 The effect of the Security Extensions

The Security Extensions can be included as part of an ARMv7-A implementation, with a VMSA. They provide two distinct 4GByte virtual memory spaces:

- a Secure virtual memory space
- a Non-secure virtual memory space.

The Secure virtual memory space is accessed by memory accesses in the Secure state, and the Non-secure virtual memory space is accessed by memory accesses in the Non-secure state.

By providing different virtual memory spaces, the Security Extensions permit memory accesses made from the Non-secure state to be distinguished from those made from the Secure state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- 命令フェッチでは、通常メモリのみにアクセスできます。デバイスメモリまたはストロングリオーダ メモリへのアクセスが実行された場合、結果は予測不能です。例えば、命令フェッチと明示的アクセスの間には順序付け要件が存在しないため、読み出しを感知するデバイスを含むメモリ領域に対して命令フェッチを実行することはできません。
- 同一のメモリ位置において次の条件が当てはまる場合、動作は予測不能です。
  - – 共有可能ノーマル、共有不可ノーマルの両方にマークされている場合
  - – 複数の異なる複数メモリタイプ（ノーマル、デバイス、ストロングリオーダ）を持つとマークされている場合
  - – 複数の異なるキャッシュ可能属性を持つとマークされている場合
  - – 共有可能デバイスメモリと共有不可デバイスメモリの両方にマークされている場合

  このようなメモリのマークにおける矛盾は、仮想アドレスから物理アドレスへのマッピングにエイリアスを用いた場合などに生じることがあります。

ARMv6 より前では、低割り込みレイテンシモードをサポートするかどうかは実装定義です。ARMv6 およびそれ以降では、低割り込みレイテンシのサポートは SCTLR.FI ビットで制御されます。マルチアクセス命令が低割り込みレイテンシ構成で正常に動作するかどうかは、実装定義です。

A3.5.8 下位互換性

ARMv6 およびそれ以降のメモリ属性は、それより前のバージョンのアーキテクチャと大きく異なっています。新しいメモリタイプ定義と以前のメモリタイプとの対応を、表 A3-9 に示します。

表 A3-9 下位互換性

以前のアーキテクチャ	ARMv6 と ARMv7 の属性
NCNB（キャッシュ不可、バッファ不可）	ストロングリオーダ
NCB（キャッシュ不可、バッファ可能）	共有可能デバイス
ライトスルー キャッシュ可能、バッファ可能	共有不可ノーマル、ライトスルー キャッシュ可能
ライトバック キャッシュ可能、バッファ可能	共有不可ノーマル、ライトバック キャッシュ可能

A3.5.9 セキュリティ拡張機能の効果

セキュリティ拡張機能は、ARMv7-A の VMSA 実装の一部として含めることができます。セキュリティ拡張機能は、2 つの 4G バイトの仮想メモリ空間を提供します。

- セキュア仮想メモリ空間
- 非セキュア仮想メモリ空間

セキュア仮想メモリ空間は、セキュア状態のメモリアクセスによってアクセスされます。非セキュア仮想メモリ空間は、非セキュア状態のメモリアクセスによってアクセスされます。

異なる仮想メモリ空間を提供することにより、セキュリティ拡張機能は、非セキュア状態からのメモリアクセスをセキュア状態からのメモリアクセスと区別することができます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.6 Access rights

ARMv7 includes additional attributes for memory regions, that enable:

- Data accesses to be restricted, based on the privilege of the access. See *Privilege level access controls for data accesses*.
- Instruction fetches to be restricted, based on the privilege of the process or thread making the fetch. See *Privilege level access controls for instruction accesses*.
- On a system that implements the Security Extensions, accesses to be restricted to memory accesses with the Secure memory attribute. See *Memory region security status* on page A3-39.

A3.6.1 Privilege level access controls for data accesses

The memory attributes can define that a memory region is:

- not accessible to any accesses
- accessible only to Privileged accesses
- accessible to Privileged and Unprivileged accesses.

The access privilege level is defined separately for explicit read and explicit write accesses. However, a system that defines the memory attributes is not required to support all combinations of memory attributes for read and write accesses.

A Privileged access is an access made during privileged execution, as a result of a load or store operation other than `LDRT`, `STRT`, `LDRBT`, `STRBT`, `LDRHT`, `STRHT`, `LDRSHT`, or `LDRSBT`.

An Unprivileged access is an access made as a result of load or store operation performed in one of these cases:

- when the processor is in an unprivileged mode
- when the processor is in any mode and the access is made as a result of a `LDRT`, `STRT`, `LDRBT`, `STRBT`, `LDRHT`, `STRHT`, `LDRSHT`, or `LDRSBT` instruction.

A Data Abort exception is generated if the processor attempts a data access that the access rights do not permit. For example, a Data Abort exception is generated if the processor is in unprivileged mode and attempts to access a memory region that is marked as only accessible to Privileged accesses.

A3.6.2 Privilege level access controls for instruction accesses

Memory attributes can define that a memory region is:

- Not accessible for execution
- Accessible for execution by Privileged processes only
- Accessible for execution by Privileged and Unprivileged processes.

To define the instruction access rights to a memory region, the memory attributes describe, separately, for the region:

- its read access rights, see *Privilege level access controls for data accesses*

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A3.6 アクセス権

ARMv7 にはメモリ領域に対する追加の属性が組み込まれており、次の処理が可能になります。

- アクセスの特権に基づいて、データアクセスを制限できます。詳細については、*データアクセスの特権レベルアクセス制御*を参照して下さい。
- プロセスの特権、またはフェッチを行うスレッドに基づいて、命令フェッチを制限できます。詳細については、*命令アクセスの特権レベルアクセス制御*を参照して下さい。
- セキュリティ拡張機能を実装しているシステムでは、セキュアメモリ属性を持つメモリアクセスのみにアクセスを制限できます。詳細については、*メモリ領域のセキュリティ状態* P. A3- 39 を参照して下さい。

A3.6.1 データアクセスの特権レベルアクセス制御

メモリ属性によって、メモリ領域に次のアクセス条件を定義できます。

- どのようなアクセスも行えない。
- 特権アクセスのみ行える。
- 特権アクセスおよび非特権アクセスの両方が行える。

アクセスの特権レベルは、明示的な読み出しおよび書き込みアクセスに対して個別に定義されます。ただし、メモリ属性を定義するシステムでは、読み出しおよび書き込みアクセスに対するメモリ属性のすべての組み合わせをサポートする必要はありません。

特権アクセスは特権モードでの実行中に、`LDRT`、`STRT`、`LDRBT`、`STRBT`、`LDRHT`、`STRHT`、`LDRSHT`、`LDRSBT` 命令を除くロード / ストア操作の結果として行われるアクセスです。

非特権アクセスは、次のいずれかの場合に、ロード / ストア操作の結果として行われるアクセスです。

- プロセッサが非特権モードにある場合
- プロセッサのモードにかかわらず、`LDRT`、`STRT`、`LDRBT`、`STRBT`、`LDRHT`、`STRHT`、`LDRSHT`、`LDRSBT` のいずれかの命令の結果としてアクセスが行われる場合

プロセッサが、アクセス権によって許可されていないデータアクセスを試みた場合は、データアボート例外が生成されます。例えば、プロセッサが非特権モードで、特権アクセスのみ可能としてマークされているメモリ領域にアクセスを試みた場合は、データアボート例外が発生します。

A3.6.2 命令アクセスの特権レベルアクセス制御

メモリ属性によって、メモリ領域について次のアクセス条件を定義できます。

- 実行アクセスは不可
- 特権プロセスのみによって、実行アクセスが可能
- 特権および非特権のプロセスによって、実行アクセスが可能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- whether it is *suitable for execution*.

For example, a region that is accessible for execution by Privileged processes only has the memory attributes:

- accessible only to Privileged read accesses
- suitable for execution.

This means there is some linkage between the memory attributes that define the accessibility of a region to explicit memory accesses, and those that define that a region can be executed.

A memory fault occurs if a processor attempts to execute code from a memory location with attributes that do not permit code execution.

A3.6.3 Memory region security status

An additional memory attribute determines whether the memory region is Secure or Non-secure in an ARMv7-A system that implements the Security Extensions. When the Security Extensions are implemented, this attribute is checked by the system hardware to ensure that a region of memory that is designated as Secure by the system hardware is not accessed by memory accesses with the Non-secure memory attribute. For more information, see *Memory region attributes* on page B3-32.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

メモリ領域への命令アクセス権を定義するため、メモリ属性には領域に対してそれぞれ独立に、次の条件が定義されます。

- 読み出しアクセス権。詳細については、*データアクセスの特権レベルアクセス制御を参照して下さい。*
- *実行に適しているかどうか*

例えば、特権プロセスによってのみ実行アクセスが可能な領域には、次のメモリ属性が指定されています。

- 特権読み出しアクセスのみでアクセスが可能
- 実行に適している

これは、明示的なメモリアクセスについて、領域へのアクセス可能性を定義するメモリ属性と、領域が実行可能であることを定義するメモリ属性との間に、何らかの関係があることを意味しています。

コード実行が許可されていない属性を持つメモリ位置からプロセッサがコードの実行を試みると、メモリフォールトが発生します。

A3.6.3 メモリ領域のセキュリティ状態

セキュリティ拡張機能を実装している ARMv7-A システムでは、追加のメモリ属性により、メモリ領域がセキュアか非セキュアかが決定されます。セキュリティ拡張機能が実装されている場合、システムハードウェアがこの属性をチェックし、セキュアとして指定されたメモリ領域が非セキュアのメモリ属性を持つメモリアクセスによってアクセスされることを防止します。詳細については、*メモリ領域の属性* P. B3-32 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.7 Virtual and physical addressing

ARMv7 provides three alternative architectural profiles, ARMv7-A, ARMv7-R and ARMv7-M. Each of the profiles specifies a different memory system. This manual describes two of these profiles:

ARMv7-A profile

The ARMv7-A memory system incorporates a *Memory Management Unit* (MMU), controlled by CP15 registers. The memory system supports virtual addressing, with the MMU performing virtual to physical address translation, in hardware, as part of program execution.

ARMv7-R profile

The ARMv7-R memory system incorporates a *Memory Protection Unit* (MPU), controlled by CP15 registers. The MPU does not support virtual addressing.

At the application level, the difference between the ARMv7-A and ARMv7-R memory systems is transparent. Regardless of which profile is implemented, an application accesses the memory map described in *Address space* on page A3-2, and the implemented memory system makes the features described in this chapter available to the application.

For a system-level description of the ARMv7-A and ARMv7-R memory models see:

- Chapter B2 *Common Memory System Architecture Features*
- Chapter B3 *Virtual Memory System Architecture (VMSA)*
- Chapter B4 *Protected Memory System Architecture (PMSA)*.

Note

This manual does not describe the ARMv7-M profile. For details of this profile see:

- *ARMv7-M Architecture Application Level Reference Manual*, for an application-level description
- *ARMv7-M Architecture Reference Manual*, for a full description.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A3.7 仮想アドレシングと物理アドレシング

ARMv7 には、ARMv7-A、ARMv7-R、ARMv7-M の 3 つの代替アーキテクチャプロファイルがあります。各プロファイルは、異なるメモリシステムを指定しています。本書では、このうち 2 つのプロファイルについて説明します。

ARMv7-A プロファイル

ARMv7-A のメモリシステムは、CP15 レジスタで制御される*メモリ管理ユニット* (MMU) を組み込んでいます。このメモリシステムは、仮想アドレシングをサポートし、MMU がプログラム実行の一部として仮想アドレスから物理アドレスへの変換をハードウェアで実行します。

ARMv7-R プロファイル

ARMv7-R のメモリシステムは、CP15 レジスタで制御される*メモリ保護ユニット* (MPU) を組み込んでいます。MPU は仮想アドレシングをサポートしません。

アプリケーションレベルにおいて、ARMv7-A と ARMv7-R のメモリシステムの違いは透過的です。どちらのプロファイルが実装されていても、アプリケーションは*アドレス空間* P. A3-2 の説明どおりにメモリマップにアクセスし、実装されているメモリシステムは本章に記載されている機能をアプリケーションに提供します。

ARMv7-A と ARMv7-R のメモリモデルに関するシステムレベルの説明については、次の各章を参照して下さい。

- 第 B2 章 *「一般的なメモリシステム アーキテクチャの機能」*
- 第 B3 章 *「仮想メモリシステム アーキテクチャ (VMSA)」*
- 第 B4 章 *「保護メモリシステム アーキテクチャ (PMSA)」*

注

本書では ARMv7-M プロファイルについて説明しません。このプロファイルの詳細については、次の資料を参照して下さい。

- 『*ARMv7-M アーキテクチャ アプリケーションレベル リファレンスマニュアル*』(アプリケーションレベルの説明)
- 『*ARMv7-M アーキテクチャ リファレンスマニュアル*』(完全な説明)

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.8 Memory access order

ARMv7 provides a set of three memory types, Normal, Device, and Strongly-ordered, with well-defined memory access properties.

The ARMv7 application-level view of the memory attributes is described in:

- *Memory types and attributes and the memory order model* on page A3-24
- *Access rights* on page A3-38.

When considering memory access ordering, an important feature of the ARMv6 memory model is the *Shareable* memory attribute, that indicates whether a region of memory can be shared between multiple processors, and therefore requires an appearance of cache transparency in the ordering model.

The key issues with the memory order model depend on the target audience:

- For software programmers, considering the model at the application level, the key factor is that for accesses to Normal memory barriers are required in some situations where the order of accesses observed by other observers must be controlled.
- For silicon implementers, considering the model at the system level, the Strongly-ordered and Device memory attributes place certain restrictions on the system designer in terms of what can be built and when to indicate completion of an access.

  Note

  Implementations remain free to choose the mechanisms required to implement the functionality of the memory model.

More information about the memory order model is given in the following subsections:

- *Reads and writes* on page A3-42
- *Ordering requirements for memory accesses* on page A3-45
- *Memory barriers* on page A3-47.

Additional attributes and behaviors relate to the memory system architecture. These features are defined in the system level section of this manual:

- Virtual memory systems based on an MMU, described in Chapter B3 *Virtual Memory System Architecture (VMSA)*.
- Protected memory systems based on an MPU, described in Chapter B4 *Protected Memory System Architecture (PMSA)*.
- Caches, described in *Caches* on page B2-3.

Note

In these system level descriptions, some attributes are described in relation to an MMU. In general, these descriptions can also be applied to an MPU based system.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.8 メモリアクセスの順序

ARMv7 では、明確なメモリアクセス プロパティを持つ、ノーマル、デバイス、ストロングリオーダの 3 つのメモリタイプのセットが用意されています。

メモリ属性の ARMv7 アプリケーションレベルのビューについては、次のセクションで説明されています。

- *メモリのタイプおよび属性とメモリオーダ モデル* P. A3- 24
- *アクセス権* P. A3- 38

メモリアクセスの順序付けについて考える場合、ARMv6 メモリモデルの重要な特徴は*共有可能*メモリ属性です。この属性は、メモリ領域が複数のプロセッサ間で共有可能か、およびその結果として順序モデルにおいて外見上のキャッシュの透過性が必要かどうかを示します。

メモリ順序モデルに関する主な考慮点は、対象読者によって次のように異なります。

- ソフトウェアプログラマの場合は、アプリケーションレベルでモデルを考えるため、主な要素はノーマルメモリに対するアクセスで、他のオブザーバによって観測されるアクセスの順序を制御することが必要とされる一部の状況においては、バリアが必要になります。
- シリコン実装者の場合は、システムレベルでモデルを考えるため、ストロングリオーダメモリおよびデバイスメモリ属性により、どのようなものが構築可能か、およびどの時点でアクセスの完了を示すかについて、システム設計者に一定の制限が課されます。

  注

  メモリモデルの機能を実装するのに必要な機構の選択は、実装者に任されています。

メモリ順序モデルの詳細については、次のサブセクションを参照して下さい。

- *読み出しと書き込み* P. A3- 42
- *メモリアクセスの順序付けの要件* P. A3- 45
- *メモリバリア* P. A3- 47

その他の属性や動作は、メモリシステム アーキテクチャに関連します。このような特徴は、本書のシステムレベルのセクションで定義されています。

- MMU をベースとする仮想メモリシステムについては、第 B3 章 *「仮想メモリシステム アーキテクチャ (VMSA)」* を参照して下さい。
- MPU をベースとする保護メモリシステムについては、第 B4 章 *「保護メモリシステム アーキテクチャ (PMSA)」* を参照して下さい。
- キャッシュについては、キャッシュ P. B2- 3 を参照して下さい。

注

これらのシステムレベルの説明において、一部の属性は MMU との関連で説明されています。これらの説明は一般に、MPU ベースのシステムにも該当します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.8.1 Reads and writes

Each memory access is either a read or a write. *Explicit* memory accesses are the memory accesses required by the function of an instruction. The following can cause memory accesses that are not explicit:

- instruction fetches
- cache loads and writebacks
- translation table walks.

Except where otherwise stated, the memory ordering requirements only apply to explicit memory accesses.

Reads

Reads are defined as memory operations that have the semantics of a load.

The memory accesses of the following instructions are reads:

- LDR, LDRB, LDRH, LDRSB, and LDRSH
- LDRT, LDRBT, LDRHT, LDRSBT, and LDRSHT
- LDREX, LDREXB, LDREXD, and LDREXH
- LDM, LDRD, POP, and RFE
- LDC, LDC2, VLDM, VLDR, VLD1, VLD2, VLD3, and VLD4
- the return of status values by STREX, STREXB, STREXD, and STREXH
- in the ARM instruction set only, SWP and SWPB
- in the Thumb instruction set only, TBB and TBH.

Hardware-accelerated opcode execution by the Jazelle extension can cause a number of reads to occur, according to the state of the operand stack and the implementation of the Jazelle hardware acceleration.

Writes

Writes are defined as memory operations that have the semantics of a store.

The memory accesses of the following instructions are Writes:

- STR, STRB, and STRH
- STRT, STRBT, and STRHT
- STREX, STREXB, STREXD, and STREXH
- STM, STRD, PUSH, and SRS
- STC, STC2, VSTM, VSTR, VST1, VST2, VST3, and VST4
- in the ARM instruction set only, SWP and SWPB.

Hardware-accelerated opcode execution by the Jazelle extension can cause a number of writes to occur, according to the state of the operand stack and the implementation of the Jazelle hardware acceleration.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A3.8.1 読み出しと書き込み

メモリへの各アクセスは、読み出しまたは書き込みのいずれかです。*明示的*メモリアクセスは、命令の機能によって要求されるメモリアクセスです。次のような操作は、明示的でないメモリアクセスを引き起こすことがあります。

- 命令フェッチ
- キャッシュのロードおよびライトバック
- 変換テーブルウォーク

特記されていない限り、メモリオーダリング要件は明示的なメモリアクセスのみに適用されます。

読み出し

読み出しは、ロードの意味を持つメモリ操作として定義されます。

次の命令のメモリアクセスは、読み出しです。

- LDR、LDRB、LDRH、LDRSB、LDRSH
- LDRT、LDRBT、LDRHT、LDRSBT、LDRSHT
- LDREX、LDREXB、LDREXD、LDREXH
- LDM、LDRD、POP、RFE
- LDC、LDC2、VLDM、VLDR、VLD1、VLD2、VLD3、VLD4
- ステータス値を返す命令：STREX、STREXB、STREXD、STREXH
- SWP、SWPB（ARM 命令セットのみ）
- TBB、TBH（Thumb 命令セットのみ）

Jazelle 拡張機能によってハードウェアで高速化されるオペコード実行では、オペランドスタックの状態と、Java ハードウェアアクセラレーションの実装によっては、大量の読み出しが発生することがあります。

書き込み

書き込みは、ストアの意味を持つメモリ操作として定義されます。

次の命令のメモリアクセスは、書き込みです。

- STR、STRB、STRH
- STRT、STRBT、STRHT
- STREX、STREXB、STREXD、STREXH
- STM、STRD、PUSH、SRS
- STC、STC2、VSTM、VSTR、VST1、VST2、VST3、VST4
- SWP、SWPB（ARM 命令セットのみ）

Jazelle 拡張機能によってハードウェアで高速化されるオペコード実行では、オペランドスタックの状態と、Jazelle ハードウェアアクセラレーションの実装によっては、大量の書き込みが発生することがあります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Synchronization primitives

Synchronization primitives must ensure correct operation of system semaphores in the memory order model. The synchronization primitive instructions are defined as those instructions that are used to ensure memory synchronization:

- LDREX, STREX, LDREXB, STREXB, LDREXD, STREXD, LDREXH, STREXH.
- SWP, SWPB. Use of these instructions is deprecated from ARMv6.

Before ARMv6, support consisted of the SWP and SWPB instructions. ARMv6 introduced new Load-Exclusive and Store-Exclusive instructions LDREX and STREX, and deprecated using the SWP and SWPB instructions. ARMv7 introduces:

- additional Load-Exclusive and Store-Exclusive instructions, LDREXB, LDREXD, LDREXH, STREXB, STREXD, and STREXH
- the Clear-Exclusive instruction CLREX
- the Load-Exclusive, Store-Exclusive and Clear-Exclusive instructions in the Thumb instruction set.

For details of the Load-Exclusive, Store-Exclusive and Clear-Exclusive instructions see *Synchronization and semaphores* on page A3-12.

The Load-Exclusive and Store-Exclusive instructions are supported to Shareable and Non-shareable memory. Non-shareable memory can be used to synchronize processes that are running on the same processor. Shareable memory must be used to synchronize processes that might be running on different processors.

Observability and completion

An *observer* is an agent in the system that can access memory. For a processor, the following mechanisms must be treated as independent observers:

- the mechanism that performs reads or writes to memory
- a mechanism that causes an instruction cache to be filled from memory or that fetches instructions to be executed directly from memory
- a mechanism that performs translation table walks.

The set of observers that can observe a memory access is defined by the system.

For all memory:

- a write to a location in memory is said to be observed by an observer when a subsequent read of the location by the same observer will return the value written by the write
- a write to a location in memory is said to be globally observed for a shareability domain when a subsequent read of the location by any observer in that shareability domain will return the value written by the write

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

同期化基本命令

同期化基本命令では、メモリ順序モデルにおいてシステムセマフォの正しい操作を保証する必要があります。次の同期化基本命令は、メモリの同期化を保証するために使用される命令として定義されます。

- LDREX、STREX、LDREXB、STREXB、LDREXD、STREXD、LDREXH、STREXH
- SWP、SWPB。ARMv6 およびそれ以降では、これらの命令の使用は推奨されていません。

ARMv6 以前では、SWP および SWPB 命令によりサポートが構成されていました。ARMv6 で新しい排他ロード命令の LDREX および排他ストア命令の STREX が導入され、SWP 命令と SWPB 命令の使用は推奨されなくなりました。ARMv7 では、次の命令が導入されています。

- 新たな排他ロード / 排他ストア命令、LDREXB、LDREXD、LDREXH、STREXB、STREXD、STREXH
- 排他クリア命令、CLREX
- Thumb 命令セットの排他ロード、排他ストア、排他クリア命令

排他ロード、排他ストア、排他クリア命令の詳細については、*同期化とセマフォ* P. A3-12 を参照して下さい。

排他ロードおよび排他ストア命令は、共有可能メモリおよび共有不可メモリに対してサポートされています。共有不可メモリは、同じプロセッサ上で実行中のプロセスの同期に使用できます。プロセスが異なるプロセッサで実行されている可能性がある場合、プロセスの同期には共有可能メモリを使用する必要があります。

観測可能性と完了

*オブザーバ*とは、メモリにアクセスできるシステム内のエージェントです。プロセッサでは、次の機構を独立したオブザーバとして扱う必要があります。

- メモリの読み出しや書き込みを実行する機構
- 命令キャッシュをメモリからフィルしたり、命令をフェッチしてメモリから直接実行したりする機構
- 変換テーブルウォークを実行する機構

メモリアクセスを観測できるオブザーバのセットは、システムによって定義されます。

すべてのメモリについて、次の規則が適用されます。

- メモリのある位置への書き込みは、オブザーバがそのメモリ位置を次に読み出したときに、その書き込みによって書き込まれた値が返される場合、そのオブザーバによって観測されると呼びます。
- メモリのある位置への書き込みは、共有可能ドメインに含まれる任意のオブザーバが次にそのメモリ位置を読み出したときに、その書き込みによって書き込まれた値が返される場合、その共有可能ドメインに関してグローバルに観測されると呼びます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- a read of a location in memory is said to be observed by an observer when a subsequent write to the location by the same observer will have no effect on the value returned by the read
- a read of a location in memory is said to be globally observed for a shareability domain when a subsequent write to the location by any observer in that shareability domain will have no effect on the value returned by the read.

Additionally, for Strongly-ordered memory:

- A read or write of a memory-mapped location in a peripheral that exhibits side-effects is said to be observed, and globally observed, only when the read or write:
  - — meets the general conditions listed
  - — can begin to affect the state of the memory-mapped peripheral
  - — can trigger all associated side effects, whether they affect other peripheral devices, processors or memory.

For all memory, the completion rules are defined as:

- A read or write is complete for a shareability domain when all of the following are true:
  - — the read or write is globally observed for that shareability domain
  - — any translation table walks associated with the read or write are complete for that shareability domain.
- A translation table walk is complete for a shareability domain when the memory accesses associated with the translation table walk are globally observed for that shareability domain, and the TLB is updated.
- A cache, branch predictor or TLB maintenance operation is complete for a shareability domain when the effects of operation are globally observed for that shareability domain and any translation table walks that arise from the operation are complete for that shareability domain.

  The completion of any cache, branch predictor and TLB maintenance operation includes its completion on all processors that are affected by both the operation and the DSB.

Side effect completion in Strongly-ordered and Device memory

The completion of a memory access in Strongly-ordered or Device memory is not guaranteed to be sufficient to determine that the side effects of the memory access are visible to all observers. The mechanism that ensures the visibility of side-effects of a memory accesses is IMPLEMENTATION DEFINED.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- メモリのある位置からの読み出しは、オブザーバがそのメモリ位置へ次に行う書き込みが、読み出される値に影響しない場合、そのオブザーバによって観測されると呼びます。
- メモリのある位置からの読み出しは、共有可能ドメインに含まれる任意のオブザーバがそのメモリ位置へ次に行う書き込みが、読み出される値に影響しない場合、その共有可能ドメインに関してグローバルに観測されると呼びます。

さらに、ストロングリオーダ メモリの場合は次の規則が適用されます。

- 副作用を示すペリフェラル内のメモリマップされた位置に対する読み出し / 書き込みは、その読み出し / 書き込みが次の条件を満たす場合にのみ観測、またはグローバルに観測されると呼びます。
  - – リストされている一般的な条件を満たしている。
  - – メモリマップされたペリフェラルの状態への影響を与え始める可能性がある。
  - – 他のペリフェラルデバイス、プロセッサ、またはメモリに対して、すべての関連する副作用をトリガする可能性がある。

すべてのメモリについて、完了規則は次のように定義されます。

- 共有可能ドメインに対する読み出し / 書き込みは、次のすべての条件が TRUE の場合に完了します。
  - – その共有可能ドメインについて、その読み出し / 書き込みがグローバルに観測される。
  - – その読み出し / 書き込みに関連する変換テーブルウォークが、その共有可能ドメインについて完了する。
- 変換テーブルウォークに関連するメモリアクセスが共有可能ドメインについてグローバルに観測される時、その共有可能ドメインについて変換テーブルウォークが完了し、TLB が更新されます。
- キャッシュ、分岐予測器、TLB の保守操作は、その操作の影響が共有可能ドメインについてグローバルに観測され、その操作によって発生した変換テーブルウォークがその共有可能ドメインについて完了した時、その共有可能ドメインについて完了します。

  キャッシュ、分岐予測器、TLB の保守操作の完了には、その操作と DSB の両方に影響を受けるすべてのプロセッサにおける完了が含まれます。

ストロングリオーダ メモリおよびデバイスメモリでの副作用の完了

ストロングリオーダ メモリまたはデバイスメモリ内でのメモリアクセスが完了しても、メモリアクセスの副作用がすべてのオブザーバに対して可視であることは保証されません。メモリアクセスの副作用の可視性を保証する機構は実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.8.2 Ordering requirements for memory accesses

ARMv7 and ARMv6 define access restrictions in the permitted ordering of memory accesses. These restrictions depend on the memory attributes of the accesses involved.

Two terms used in describing the memory access ordering requirements are:

Address dependency

An address dependency exists when the value returned by a read access is used to compute the virtual address of a subsequent read or write access. An address dependency exists even if the value read by the first read access does not change the virtual address of the second read or write access. This might be the case if the value returned is masked off before it is used, or if it has no effect on the predicted address value for the second access.

Control dependency

A control dependency exists when the data value returned by a read access is used to determine the condition code flags, and the values of the flags are used for condition code checking to determine the address of a subsequent read access. This address determination might be through conditional execution, or through the evaluation of a branch.

Figure A3-4 on page A3-46 shows the memory ordering between two explicit accesses A1 and A2, where A1 occurs before A2 in program order. The symbols used in the figure are as follows:

< Accesses must be observed in program order, that is, A1 must be observed before A2.

- Accesses can be observed in any order, provided that the requirements of uniprocessor semantics, for example respecting dependencies between instructions in a single processor, are maintained.

  The following additional restrictions apply to the ordering of memory accesses that have this symbol:

  - If there is an address dependency then the two memory accesses are observed in program order by any observer in the common shareability domain of the two accesses.

    This ordering restriction does not apply if there is only a control dependency between the two read accesses.

    If there is both an address dependency and a control dependency between two read accesses the ordering requirements of the address dependency apply.
  - If the value returned by a read access is used as data written by a subsequent write access, then the two memory accesses are observed in program order.
  - It is impossible for an observer in the shareability domain of a memory location to observe a write access to that memory location if that location would not be written to in a sequential execution of a program.
  - It is impossible for an observer in the shareability domain of a memory location to observe a write value written to that memory location if that value would not be written in a sequential execution of a program.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.8.2 メモリアクセスの順序付けの要件

ARMv7 と ARMv6 では、メモリアクセスの許可された順序付けにおけるアクセスの制約事項が定義されています。これらの制約事項は、該当するアクセスのメモリ属性に依存します。

メモリアクセスの順序付けの要件を説明するために、次の 2 つの用語が使用されます。

アドレス依存関係

アドレス依存関係は、読み出しアクセスによって返された値が、次の読み出し / 書き込みアクセスの仮想アドレスを計算するために使用される場合に存在します。最初の読み出しアクセスによって読み出された値が 2 番目の読み出し / 書き込みアクセスの仮想アドレスを変更しない場合でも、アドレス依存関係は存在します。これは、返された値が使用される前にマスクオフされるか、または 2 番目のアクセスの予測されたアドレス値に影響を与えない場合です。

制御依存関係

制御依存関係は、読み出しアクセスによって返されたデータ値が条件コードフラグの決定に使用され、そのフラグの値を条件コードのチェックに使用して次の読み出しアクセスのアドレスが決定される場合に存在します。このアドレス決定は、条件付き実行または分岐の評価を通して行われます。

P. A3- 46 の図 A3- 4 は、2 回の明示的アクセス A1 と A2 のメモリ順序を示したものです（A1 はプログラムで A2 よりも前にあります）。この図では、次の記号を使用しています。

< アクセスはプログラムの順序で観測される必要があります。つまり、A1 は A2 より前に観測される必要があります。

– ユニプロセッサセマンティクスの要件が満たされている場合、例えば単一プロセッサ内において命令間の依存関係が順守されている場合、アクセスはどのような順序で観測されてもかまいません。

  次の追加の制約事項は、この記号を持つメモリアクセスの順序付けに適用されます。

  - アドレス依存関係が存在する場合、2 つのメモリアクセスは、2 つのアクセスに共通の共有可能ドメイン内にあるどのオブザーバにもプログラムの順序で観測されます。

    この順序付けの制約事項は、2 つの読み出しアクセスの間に制御依存関係のみが存在する場合には適用されません。

    2 つの読み出しアクセス間にアドレス依存関係と制御依存関係の両方が存在する場合は、アドレス依存関係の順序付け要件が適用されます。
  - 読み出しアクセスによって返された値が、次の書き込みアクセスによって書き込まれるデータとして使用される場合、2 つのメモリアクセスはプログラムの順序で観測されます。
  - メモリ位置にプログラムのシーケンシャルな実行で書き込みが行われない場合、メモリ位置の共有可能ドメインにあるオブザーバが、そのメモリ位置への書き込みアクセスを観測することはできません。
  - プログラムのシーケンシャルな実行で値が書き込まれない場合、メモリ位置の共有可能ドメインにあるオブザーバが、そのメモリ位置に対する書き込み値を観測することはできません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- It is impossible for an observer in the shareability domain of a memory location to observe two reads to the same memory location performed by the same observer in an order that would not occur in a sequential execution of a program.

In Figure A3-4, an access refers to a read or a write access to the specified memory type. For example, *Device access, Non-shareable* refers to a read or write access to Non-shareable Device memory.

A1 \ A2	Normal access	Device access		Strongly-ordered access
		Non-shareable	Shareable	
Normal access	-	-	-	-
Device access, Non-shareable	-	<	-	<
Device access, Shareable	-	-	<	<
Strongly-ordered access	-	<	<	<

Figure A3-4 Memory ordering restrictions

There are no ordering requirements for implicit accesses to any type of memory.

Program order for instruction execution

The program order of instruction execution is the order of the instructions in the control flow trace.

Explicit memory accesses in an execution can be either:

Strictly Ordered
Denoted by <. Must occur strictly in order.

Ordered Denoted by <=. Can occur either in order or simultaneously.

Load/store multiple instructions, such as `LDM`, `LDRD`, `STM`, and `STRD`, generate multiple word accesses, each of which is a separate access for the purpose of determining ordering.

The rules for determining program order for two accesses A1 and A2 are:

If A1 and A2 are generated by two different instructions:

- A1 < A2 if the instruction that generates A1 occurs before the instruction that generates A2 in program order
- A2 < A1 if the instruction that generates A2 occurs before the instruction that generates A1 in program order.

If A1 and A2 are generated by the same instruction:

- If A1 and A2 are the load and store generated by a `SWP` or `SWPB` instruction:
  - A1 < A2 if A1 is the load and A2 is the store
  - A2 < A1 if A2 is the load and A1 is the store.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- メモリ位置の共有可能ドメインにあるオブザーバは、プログラムのシーケンシャルな実行によっては発生しえない順序で同じメモリ位置に対して同じオブザーバによって実行された、2 つの読み出しを観測することができません。

図 A3-4 で、アクセスとは指定されたメモリタイプへの読み出し / 書き込みアクセスを意味しています。例えば、*デバイスアクセス、共有不可*は、共有不可デバイスメモリへの読み出し / 書き込みアクセスの意味です。

A1 \ A2	通常アクセス	デバイスアクセス		ストロングリオーダアクセス
		共有不可	共有可	
通常アクセス	-	-	-	-
デバイスアクセス、共有不可	-	<	-	<
デバイスアクセス、共用可	-	-	<	<
ストロングリオーダ アクセス	-	<	<	<

図 A3-4 メモリ順序の制約事項

どのタイプのメモリの場合でも、暗黙アクセスの順序付け要件は存在しません。

命令実行のプログラム順序

命令実行のプログラム順序は、制御フロートレース内の命令の順序になります。

実行における明示的メモリアクセスは、次のいずれかです。

厳密な順序付け
< で示されます。厳密に指定の順序で行われる必要があります。

順序付け <= で示されます。指定の順序で、または同時に行うことができます。

`LDM`、`LDRD`、`STM`、`STRD` などの複数ロード / ストア命令は、複数のワードアクセスを生成しますが、これらのアクセスは順序を決定する目的においては別々のアクセスと見なされます。

2 つのアクセス A1 と A2 について、プログラム順序を決定するための規則は次のとおりです。

A1 と A2 が 2 つの異なる命令により生成されている場合

- プログラム順序で、A1 を生成する命令が A2 を生成する命令よりも前にある場合、A1 < A2
- プログラム順序で、A2 を生成する命令が A1 を生成する命令よりも前にある場合、A2 < A1

A1 と A2 が同じ命令により生成されている場合

- A1 と A2 が `SWP` または `SWPB` 命令により生成されたロードとストアの場合
  - A1 がロードで A2 がストアであれば、A1 < A2
  - A2 がロードで A1 がストアであれば、A2 < A1

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- In these descriptions:
  - — an *LDM-class* instruction is any form of LDM, LDMDA, LDMDB, LDMIB, or POP instruction
  - — an *LDC-class* instruction is an LDC, VLDM, or VLDR instruction
  - — an *STM-class* instruction is any form of STM, STMDA, STMDB, STMIB, or PUSH instruction
  - — an *STC-class* instruction is an STC, VSTM, or VSTR instruction.

  If A1 and A2 are two word loads generated by an LDC-class or LDM-class instruction, or two word stores generated by an STC-class or STM-class instruction, excluding LDM-class and STM-class instructions with a register list that includes the PC:
  - — A1 <= A2 if the address of A1 is less than the address of A2
  - — A2 <= A1 if the address of A2 is less than the address of A1.

  If A1 and A2 are two word loads generated by an LDM-class instruction with a register list that includes the PC or two word stores generated by an STM-class instruction with a register list that includes the PC, the program order of the memory accesses is not defined.
- If A1 and A2 are two word loads generated by an LDRD instruction or two word stores generated by an STRD instruction, the program order of the memory accesses is not defined.
- If A1 and A2 are load or store accesses generated by Advanced SIMD element or structure load/store instructions, the program order of the memory accesses is not defined.
- For any instruction or operation not explicitly mentioned in this section, if the single-copy atomicity rules described in *Single-copy atomicity* on page A3-27 mean the operation becomes a sequence of accesses, then the time-ordering of those accesses is not defined.

A3.8.3 Memory barriers

Memory barrier is the general term applied to an instruction, or sequence of instructions, used to force synchronization events by a processor with respect to retiring load/store instructions. The ARM architecture defines a number of memory barriers that provide a range of functionality, including:

- ordering of issued load/store instructions to the programmers' model
- completion of preceding load/store instructions to the programmers' model
- flushing of any instructions prefetched before the memory barrier operation.

ARMv7 and ARMv6 require three explicit memory barriers to support the memory order model described in this chapter. In ARMv7 the memory barriers are provided as instructions that are available in the ARM and Thumb instruction sets, and in ARMv6 the memory barriers are performed by CP15 register writes. The three memory barriers are:

- Data Memory Barrier, see *Data Memory Barrier (DMB)* on page A3-48
- Data Synchronization Barrier, see *Data Synchronization Barrier (DSB)* on page A3-49
- Instruction Synchronization Barrier, see *Instruction Synchronization Barrier (ISB)* on page A3-49.

Depending on the synchronization needed, a program might use memory barriers on their own, or it might use them in conjunction with cache and memory management maintenance operations that are only available in privileged modes.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- この説明で、命令の定義は次のとおりです。
  - – *LDM クラス*命令とは、あらゆる形式の LDM、LDMDA、LDMDB、LDMIB、POP 命令です。
  - – *LDC クラス*命令とは、LDC、VLDM、VLDR のいずれかの命令です。
  - – *STM クラス*命令とは、あらゆる形式の STM、STMDA、STMDB、STMIB、PUSH 命令です。
  - – *STC クラス*命令とは、STC、VSTM、VSTR のいずれかの命令です。

  A1 と A2 が、LDC クラスまたは LDM クラスの命令によって生成された 2 ワードロード、あるいは STC クラスまたは STM クラスの命令によって生成された 2 ワードストアであり、かつレジスタリストに PC が含まれる LDM クラスおよび STM クラス命令以外の場合
  - – A1 のアドレスが A2 のアドレスよりも小さい場合、A1 <= A2
  - – A2 のアドレスが A1 のアドレスよりも小さい場合、A2 <= A1

  A1 と A2 が、レジスタリストに PC を含む LDM クラス命令により生成された 2 ワードロード、またはレジスタリストに PC を含む STM クラス命令により生成された 2 ワードストアの場合、メモリアクセスのプログラム順序は定義されません。
- A1 と A2 が LDRD 命令によって生成される 2 ワードロードか、STRD 命令によって生成される 2 ワードストアである場合、メモリアクセスのプログラム順序は定義されません。
- A1 と A2 が、アドバンスト SIMD のエレメントまたは構造のロード / ストア命令によって生成されるロード / ストアアクセスの場合、メモリアクセスのプログラム順序は定義されません。
- このセクションで明示的に説明されていない命令または操作については、*シングルコピーのアトミック性* P. A3-27 で説明されているシングルコピーのアトミック性の規則により操作がアクセスのシーケンスになることを意味する場合、それらのアクセスの時間的な順序は定義されません。

A3.8.3 メモリバリア

*メモリバリア*は、命令または命令のシーケンスに適用される一般的な用語で、ロード / ストア命令の終了に関して、プロセッサによる同期イベントを強制的に発生させるために使用されます。ARM アーキテクチャでは、次のような多様な機能を持つ多くのメモリバリアを定義しています。

- プログラマモデルに対する、発行されたロード / ストア命令の順序付け
- プログラマモデルに対する、前のロード / ストア命令の完了
- メモリバリア操作の前にプリフェッチされた命令のフラッシュ

ARMv7 と ARMv6 では、本章で説明されているメモリ順序モデルをサポートするため、3 つの明示的なメモリバリアが必要です。ARMv7 では、ARM 命令セットと Thumb 命令セットで使用できる命令としてメモリバリアが提供されています。ARMv6 では、CP15 レジスタへの書き込みによってメモリバリアが実行されます。3 つのメモリバリアは次のとおりです。

- データ メモリバリア。詳細については、*データ メモリバリア (DMB)* P. A3-48 を参照して下さい。
- データ同期バリア。詳細については、*データ同期バリア (DSB)* P. A3-49 を参照して下さい。
- 命令同期バリア。詳細については、*命令同期バリア (ISB)* P. A3-49 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The DMB and DSB memory barriers affect reads and writes to the memory system generated by load/store instructions and data or unified cache maintenance operations being executed by the processor. Instruction fetches or accesses caused by a hardware translation table access are not explicit accesses.

Data Memory Barrier (DMB)

The `DMB` instruction is a data memory barrier. The processor that executes the `DMB` instruction is referred to as the executing processor, Pe. The `DMB` instruction takes the *required shareability domain* and *required access types* as arguments. If the required shareability is *Full system* then the operation applies to all observers within the system.

A DMB creates two groups of memory accesses, Group A and Group B:

Group A Contains:

- All explicit memory accesses of the required access types from observers in the same required shareability domain as Pe that are observed by Pe before the `DMB` instruction. These accesses include any accesses of the required access types and required shareability domain performed by Pe.
- All loads of required access types from observers in the same required shareability domain as Pe that have been observed by any given observer, Py, in the same required shareability domain as Pe before Py has performed a memory access that is a member of Group A.

Group B Contains:

- All explicit memory accesses of the required access types by Pe that occur in program order after the `DMB` instruction.
- All explicit memory accesses of the required access types by any given observer Px in the same required shareability domain as Pe that can only occur after Px has observed a store that is a member of Group B.

Any observer with the same required shareability domain as Pe observes all members of Group A before it observes any member of Group B to the extent that those group members are required to be observed, as determined by the shareability and cacheability of the memory locations accessed by the group members. Where members of Group A and Group B access the same memory-mapped peripheral, all members of Group A will be visible at the memory-mapped peripheral before any members of Group B are visible at that peripheral.

Note

- A memory access might be in neither Group A nor Group B. The DMB does not affect the order of observation of such a memory access.
- The second part of the definition of Group A is recursive. Ultimately, membership of Group A derives from the observation by Py of a load before Py performs an access that is a member of Group A as a result of the first part of the definition of Group A.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

必要な同期化に応じ、プログラムはメモリバリアを単独で、あるいは特権モードでのみ使用可能なキャッシュおよびメモリ管理保守操作と組み合わせて使用することがあります。

DMB および DSB メモリバリアは、ロード / ストア命令によって生成されるメモリシステムへの読み出し / 書き込み、およびプロセッサが実行するデータまたは統一キャッシュ保守操作に影響を与えます。ハードウェア変換テーブルアクセスによって生じる命令のフェッチまたはアクセスは明示的なアクセスではありません。

データ メモリバリア (DMB)

`DMB` 命令は、データ メモリバリアです。`DMB` 命令を実行するプロセッサは、実行中プロセッサ (Pe) として示されます。`DMB` 命令は、*共有可能必須ドメイン*および*必要アクセス タイプ*を引数として使用します。共有可能必須が*フルシステム*の場合、この操作はシステム内のすべてのオブザーバに適用されます。

DMB は、グループ A およびグループ B の 2 つのグループのメモリアクセスを作成します。

グループ A 次のものから構成されます。

- Pe と同じ共有可能必須ドメイン内のオブザーバによって行われ、`DMB` 命令の前に Pe によって観測される、必要アクセスタイプのすべての明示的メモリアクセス。これには、Pe によって実行される必要アクセスタイプおよび共有可能必須ドメインのすべてのアクセスが含まれます。
- Pe と同じ共有可能必須ドメイン内のオブザーバによる必須アクセスタイプのすべてのロードで、Pe と同じ共有可能必須ドメイン内の任意のオブザーバ Py によって、Py がグループ A のメンバであるメモリアクセスを実行する前に観測されるもの。

グループ B 次のものから構成されます。

- Pe による必要アクセスタイプのすべての明示的メモリアクセスで、プログラム順序で `DMB` 命令の後に発生するもの。
- Pe と同じ共有可能必須ドメイン内の任意のオブザーバ Px による、必要アクセスタイプのすべての明示的メモリアクセスで、Px がグループ B のメンバであるストアを観測した後にのみ発生可能なもの。

Pe と同じ共有可能必須ドメインを持つ任意のオブザーバは、各グループメンバがアクセスするメモリ位置の共有可能性およびキャッシュ可能性から判断してグループメンバを観測する必要がある限り、グループ B のメンバを観測する前にグループ A のすべてのメンバを観測します。グループ A およびグループ B のメンバは同じメモリマップされたペリフェラルにアクセスしますが、グループ A のすべてのメンバは、グループ B のいずれかのメンバがそのメモリマップされたペリフェラルから可視になる前に、そのペリフェラルから可視になります。

注

- メモリアクセスは、グループ A とグループ B のいずれにも属さない可能性があります。DMB は、そのようなメモリアクセスの観測の順序には影響を与えません。
- グループ A の定義の 2 番目の部分は再帰的です。最終的に、グループ A のメンバ構成は、グループ A の定義の最初の部分によりグループ A のメンバとして定義されるアクセスを Py が実行する前に、Py がロードを観測することから導き出されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- The second part of the definition of Group B is recursive. Ultimately, membership of Group B derives from the observation by any observer of an access by Pe that is a member of Group B as a result of the first part of the definition of Group B.

DMB only affects memory accesses. It has no effect on the ordering of any other instructions executing on the processor.

For details of the DMB instruction in the Thumb and ARM instruction sets see *DMB* on page A8-90.

Data Synchronization Barrier (DSB)

The DSB instruction is a special memory barrier, that synchronizes the execution stream with memory accesses. The DSB instruction takes the *required shareability domain* and *required access types* as arguments. If the required shareability is *Full system* then the operation applies to all observers within the system.

A DSB behaves as a DMB with the same arguments, and also has the additional properties defined here.

A DSB completes when both:

- all explicit memory accesses that are observed by Pe before the DSB is executed, are of the required access types, and are from observers in the same required shareability domain as Pe, are complete for the set of observers in the required shareability domain
- all cache, branch predictor, and TLB maintenance operations issued by Pe before the DSB are complete for the required shareability domain.

In addition, no instruction that appears in program order after the DSB instruction can execute until the DSB completes.

For details of the DSB instruction in the Thumb and ARM instruction sets see *DSB* on page A8-92.

Note

Historically, this operation was referred to as Drain Write Buffer or Data Write Barrier (DWB). From ARMv6, these names and the use of DWB were deprecated in favor of the new Data Synchronization Barrier name and DSB abbreviation. DSB better reflects the functionality provided from ARMv6, because DSB is architecturally defined to include all cache, TLB and branch prediction maintenance operations as well as explicit memory operations.

Instruction Synchronization Barrier (ISB)

An ISB instruction flushes the pipeline in the processor, so that all instructions that come after the ISB instruction in program order are fetched from cache or memory only after the ISB instruction has completed. Using an ISB ensures that the effects of context altering operations executed before the ISB are visible to the instructions fetched after the ISB instruction. Examples of context altering operations that require the insertion of an ISB instruction to ensure the operations are complete are:

- cache, TLB, and branch predictor maintenance operations
- changes to the CP14 and CP15 registers.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- グループ B の定義の 2 番目の部分は再帰的です。最終的に、グループ B のメンバ構成は、グループ B の定義の最初の部分によりグループ B のメンバとして定義される Pe によるアクセスが、任意のオブザーバによって観測されることから導き出されます。

DMB はメモリアクセスにのみ影響を与えます。プロセッサ上で実行されている他の命令の順序には影響を与えません。

Thumb 命令セットおよび ARM 命令セットにおける DMB 命令の詳細については、*DMB* P. A8-90 を参照して下さい。

データ同期バリア (DSB)

DSB 命令は特別なメモリバリアで、実行ストリームをメモリアクセスと同期します。DSB 命令は、*共有可能必須ドメイン*および*必要アクセス タイプ*を引数として使用します。共有可能必須が*フルシステム*の場合、この操作はシステム内のすべてのオブザーバに適用されます。

DSB は同じ引数を持つ DMB として動作し、ここで定義されている追加のプロパティも持っています。

DSB 命令は、次の両方の条件が満たされた場合に完了します。

- DSB が実行される前に Pe によって観測され、必要アクセスタイプを持ち、Pe と同じ共有可能必須ドメイン内のオブザーバによって行われ、その共有可能必須ドメイン内のオブザーバのセットに対して完了する、すべての明示的メモリアクセス
- 共有可能必須ドメインについて DSB が完了する前に Pe によって発行される、すべてのキャッシュ、分岐予測器、TLB の保守操作

さらに、プログラム順序で DSB 命令の後に出現する命令は、DSB 命令が完了するまで実行されません。

Thumb 命令セットおよび ARM 命令セットにおける DSB 命令の詳細については、*DSB* P. A8-92 を参照して下さい。

注

従来は、この操作はライトバッファ ドレインまたはデータ書き込みバリア (DWB) と呼ばれていました。ARMv6 から、これらの名称と DWB の略称は推奨されなくなり、新しいデータ同期バリアという名称と DSB という略称が推奨されるようになりました。DSB の名称は、すべてのキャッシュ、TLB、分岐予測の保守操作に加え、明示的なメモリ操作を含むとアーキテクチャで定義されているため、ARMv6 の提供する機能をより適切に反映しています。

命令同期バリア (ISB)

ISB 命令は、プロセッサ内のパイプラインをフラッシュし、プログラム順序で ISB 命令の後に出現するすべての命令が、ISB 命令の完了後にのみキャッシュまたはメモリからフェッチされるようにします。ISB 命令を使用することにより、ISB 命令の前に実行されたコンテキスト変更操作

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In addition, any branches that appear in program order after the ISB instruction are written into the branch prediction logic with the context that is visible after the ISB instruction. This is needed to ensure correct execution of the instruction stream.

Any context altering operations appearing in program order after the ISB instruction only take effect after the ISB has been executed.

For details of the ISB instruction in the Thumb and ARM instruction sets see *ISB* on page A8-102.

Pseudocode details of memory barriers

The following types define the required shareability domains and required access types used as arguments for DMB and DSB instructions:

```
enumeration MBReqDomain {MBReqDomain_FullSystem,
                         MBReqDomain_OuterShareable,
                         MBReqDomain_InnerShareable,
                         MBReqDomain_Nonshareable};

enumeration MBReqTypes {MBReqTypes_All, MBReqTypes_Writes};
```

The following procedures perform the memory barriers:

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types)
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types)
InstructionSynchronizationBarrier()
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

の作用が、ISB 命令後にフェッチされた命令から可視であることが保証されます。操作が完了したことを保証するため、ISB 命令の挿入を必要とする可能性があるコンテキスト変更操作の例として、次のものが挙げられます。

- キャッシュ、TLB、分岐予測器の保守操作
- CP14 および CP15 レジスタの変更

さらに、プログラム順序で ISB 命令の後に出現するすべての分岐命令は、ISB 命令後に可視になるコンテキストとともに分岐予測ロジックに書き込まれます。これは、命令ストリームの正確な実行を保証するために必要になります。

プログラム順序で ISB 命令の後に出現するすべてのコンテキスト変更操作は、ISB 命令が実行された後でのみ有効になります。

Thumb 命令セットおよび ARM 命令セットにおける ISB 命令の詳細については、*ISB* P. A8- 102 を参照して下さい。

メモリバリアの擬似コードの詳細

次の型は、DMB および DSB 命令の引数として使用される、共有可能必須ドメインと必要アクセスタイプを定義しています。

```
enumeration MBReqDomain {MBReqDomain_FullSystem,
                         MBReqDomain_OuterShareable,
                         MBReqDomain_InnerShareable,
                         MBReqDomain_Nonshareable};

enumeration MBReqTypes {MBReqTypes_All, MBReqTypes_Writes};
```

次のプロジージャにより、メモリバリアが実行されます。

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types)
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types)
InstructionSynchronizationBarrier()
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.9 Caches and memory hierarchy

The implementation of a memory system depends heavily on the microarchitecture and therefore the details of the system are IMPLEMENTATION DEFINED. ARMv7 defines the application level interface to the memory system, and supports a hierarchical memory system with multiple levels of cache. This section provides an application level view of this system. It contains the subsections:

- *Introduction to caches*
- *Memory hierarchy* on page A3-52
- *Implication of caches for the application programmer* on page A3-52
- *Preloading caches* on page A3-54.

A3.9.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- main memory address information, commonly known as a *tag*
- the associated data.

Caches are used to increase the average speed of a memory access. Cache operation takes account of two principles of locality:

Spatial locality

An access to one location is likely to be followed by accesses to adjacent locations. Examples of this principle are:

- sequential instruction execution
- accessing a data structure.

Temporal locality

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a code loop

To minimize the quantity of control information stored, the spatial locality property is used to group several locations together under the same tag. This logical block is commonly known as a cache line. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a cache hit, and other accesses are called cache misses.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the processor wants to access a cacheable location, the cache is checked. If the access is a cache hit, the access occurs in the cache, otherwise a location is allocated and the cache line loaded from memory. Different cache topologies and access policies are possible, however, they must comply with the memory coherency model of the underlying architecture.

Caches introduce a number of potential problems, mainly because of:

- Memory accesses occurring at times other than when the programmer would normally expect them
- There being multiple physical locations where a data item can be held

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.9 キャッシュとメモリ階層

メモリシステムの実装は、マイクロアーキテクチャに強く依存しているため、システムの詳細は実装定義です。ARMv7 は、メモリシステムに対するアプリケーションレベルのインタフェースを定義し、複数のレベルのキャッシュを持つ階層型メモリシステムをサポートしています。本セクションでは、アプリケーションレベルから見たこのシステムについて説明します。このセクションは次のサブセクションで構成されています。

- *キャッシュの概要*
- *メモリの階層* P. A3-52
- *アプリケーションプログラマへのキャッシュの影響* P. A3-52
- *キャッシュのプリロード* P. A3-54

A3.9.1 キャッシュの概要

キャッシュとは、多数のエントリを含む高速メモリのブロックです。各エントリは次の要素で構成されます。

- 一般にタグと呼ばれるメインメモリのアドレス情報
- 関連するデータ

キャッシュの目的は、メモリアクセスの平均速度を上げることです。キャッシュは、次の 2 つの局所性の原則に基づいて動作します。

空間的局所性

1 つの位置へのアクセスの後には、隣接位置へのアクセスがある可能性が高いという原則です。この原則の例には、次のようなものがあります。

- シーケンシャルな命令実行
- データ構造へのアクセス

時間的局所性

メモリ領域へのアクセスは、短時間内に繰り返される可能性が高いという原則です。この原則の例には、コードループの実行があります。

格納される制御情報の数を最小限にするため、空間的局所性プロパティを使用して、同じタグのもとで複数の位置がグループ化されます。この論理ブロックは、一般にキャッシュラインと呼ばれます。データがキャッシュにロードされると、次のロード / ストアのアクセスタイムが短縮され、全体のパフォーマンスが向上します。すでにキャッシュにある情報へのアクセスはキャッシュヒットと呼ばれ、他のアクセスはキャッシュミスと呼ばれます。

通常、キャッシュは自己管理され、自動的に更新されます。プロセッサがキャッシュ可の位置にアクセスする場合、常にキャッシュがチェックされます。アクセスがキャッシュヒットの場合、アクセスはキャッシュ内で発生し、そうでない場合は位置が割り当てられ、キャッシュラインがメモリからロードされます。別のキャッシュトポロジおよびアクセスポリシーも使用可能ですが、基礎となるアーキテクチャのメモリコヒーレンシ モデルに準拠している必要があります。

キャッシュによりいくつかの潜在的な問題が生じますが、主な原因は次のとおりです。

- プログラマが通常予期していない時期にメモリアクセスが発生する可能性がある。
- データ項目が保持される物理的位置が複数になる。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.9.2 Memory hierarchy

Memory close to a processor has very low latency, but is limited in size and expensive to implement. Further from the processor it is easier to implement larger blocks of memory but these have increased latency. To optimize overall performance, an ARMv7 memory system can include multiple levels of cache in a hierarchical memory system. Figure A3-5 shows such a system, in an ARMv7-A implementation of a VMSA, supporting virtual addressing.

Figure A3-5 Multiple levels of cache in a memory hierarchy

Note

In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the processor, as shown in Figure A3-5.

A3.9.3 Implication of caches for the application programmer

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

- when memory locations are updated by other agents in the system
- when memory updates made from the application code must be made visible to other agents in the system.

For example:

- In a system with a DMA controller that reads memory locations that are held in the data cache of a processor, a breakdown of coherency occurs when the processor has written new data in the data cache, but the DMA controller reads the old data held in memory.
- In a Harvard architecture of caches, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.9.2 メモリの階層

プロセッサに近いメモリは、非常にレイテンシが低くなりますが、サイズが限られ、実装コストが高くなります。プロセッサからの距離が遠くなると、大きなメモリブロックの実装が容易となりますが、レイテンシは高くなります。全体的な性能を最適化するため、ARMv7 のメモリシステムには複数のレベルのキャッシュを階層型メモリシステムとして含めることができます。図 A3-5 は、ARMv7-A の VMSA 実装におけるそのようなシステムで、仮想アドレシングをサポートしています。

図 A3-5 メモリ階層における複数レベルのキャッシュ

注

本書では、階層型メモリシステムにおけるレベル 1 が、最もプロセッサに近いレベルを意味します。図 A3-5 を参照して下さい。

A3.9.3 アプリケーションプログラマへのキャッシュの影響

通常の動作において、キャッシュの大部分はアプリケーションプログラマに認識されません。しかし、キャッシュのコヒーレンシが破綻した場合には明確に認識されます。このような破綻は、次の場合に発生します。

- メモリ位置がシステム内の他のエージェントによって更新された場合
- アプリケーションコードから実行されたメモリ更新を、システム内の他のエージェントに対して可視にする必要がある場合

次に例を示します。

- DMA コントローラを使用してプロセッサのデータキャッシュ内に保持されているメモリ位置を読み出すシステムでは、プロセッサがデータキャッシュに新規データを書き込んだにもかかわらず、DMA コントローラがメモリ内に保持されている古いデータを読み出した場合に、コヒーレンシの破綻が発生します。
- ハーバードアーキテクチャのキャッシュでは、命令キャッシュとデータキャッシュが独立しているため、新しい命令データがデータキャッシュに書き込まれたにもかかわらず、命令キャッシュが古い命令データを保持している場合にコヒーレンシの破綻が発生します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Data coherency issues

You can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. You can achieve this by:
  - — using Non-cacheable or, in some cases, Write-Through Cacheable memory for the caches
  - — not enabling caches in the system.
- By using cache maintenance operations to manage the coherency issues in software, see *Cache maintenance functionality* on page B2-9. Many of these operations are only available to system software.
- By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see *Non-shareable Normal memory* on page A3-30 and *Shareable, Inner Shareable, and Outer Shareable Normal memory* on page A3-30.

  The performance of these hardware coherency mechanisms is highly implementation specific. In some implementations the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.

Instruction coherency issues

How far ahead of the current point of execution instructions are prefetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- the processor might have fetched the instructions from memory at any time since the last ISB, exception entry or exception return executed by that processor
- any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being refetched from memory

In addition, the ARM architecture does not require the hardware to ensure coherency between instruction caches and memory, even for regions of memory with Shareable attributes. This means that for cacheable regions of memory, an instruction cache can hold instructions that were fetched from memory before the last ISB, exception entry or exception return.

If software requires coherency between instruction execution and memory, it must manage this coherency using the ISB and DSB memory barriers and cache maintenance operations, see *Ordering of cache and branch predictor maintenance operations* on page B2-21. Many of these operations are only available to system software.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

データコヒーレンシの問題

キャッシュのデータコヒーレンシは、次のような方法で保証できます。

- コヒーレンシの問題が生じる可能性のある状況ではキャッシュを使用しない。このためには、次のような方針を採用します。
  - – キャッシュ不可、あるいは場合によってライトスルーキャッシュ可能メモリをキャッシュの代わりに使用する
  - – システム内のキャッシュを許可しない
- キャッシュ保守操作を使用して、ソフトウェアでコヒーレンシ問題を管理する。キャッシュ*保守機能* P. B2-9 を参照して下さい。このような操作の多くはシステムソフトウェアしか実行できません。
- ハードウェアのコヒーレンシ機構を使用し、異なる共有可能ドメインにあるオブザーバによってキャッシュ可能な位置を保持するメモリに対して、データアクセスのコヒーレンシを保証する。*共有不可のノーマルメモリ* P. A3-30 および*共有可能*、*内部共有可能*、*外部共有可能のノーマルメモリ* P. A3-30 を参照して下さい。

  このようなハードウェアのコヒーレンシ機構の性能は、実装によって大きく異なります。実装によっては、このような機構により、キャッシュ共有可能な位置の能力が抑制されることもあります。また、キャッシュコヒーレンシ ハードウェアが、共有可能ドメイン内のオブザーバ間のコヒーレンシを管理しつつ、キャッシュにデータを保持できる実装もあります。

命令コヒーレンシの問題

現在の実行時点からどれだけ先の命令をプリフェッチするかは、実装定義です。いくつ先の命令をプリフェッチするかは、固定することも、動的に変化させることもできます。また、実行可能な以後の実行パスのどれかを選択することも、すべてを選択することもできます。すべてのメモリのタイプについて、次の条件が適用されます。

- プロセッサは、最後に ISB、例外開始、または例外からの復帰を実行した後で、どの時点でもメモリから命令をフェッチした可能性があります。
- この方法でフェッチされた命令は、プログラムを実行する際の必要に応じ、メモリから再フェッチされることなく複数回実行される可能性があります。

さらに、ARM アーキテクチャは、共有可能属性を持つメモリ領域についても、ハードウェアで命令キャッシュとメモリの間のコヒーレンシを保証することを要求していません。つまり、メモリのキャッシュ可能領域では、最後に実行された ISB、例外開始、または例外からの復帰より前にメモリからフェッチした命令が、命令キャッシュに保持されている可能性があります。

ソフトウェアで、命令実行とメモリの間のコヒーレンシが要求される場合、ソフトウェアは ISB および DSB メモリバリアとキャッシュ保守操作を使ってこのコヒーレンシを管理する必要があります。キャッシュの*保守操作と分岐予測器の保守操作との順序付け* P. B2-21 を参照して下さい。このような操作の多くはシステムソフトウェアしか実行できません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A3.9.4 Preloading caches

The ARM architecture provides memory system hints PLD (Preload Data) and PLI (Preload Instruction) to permit software to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if and when they do occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations will use this information to bring the data or instruction locations into caches that have faster access times than normal memory.

The Preload instructions are hints, and so implementations can treat them as NOPs without affecting the functional behavior of the device. The instructions do not generate synchronous Data Abort exceptions, but the memory system operations might, under exceptional circumstances, generate asynchronous aborts. For more information, see *Data Abort exception* on page B1-55.

Hardware implementations can provide other implementation-specific mechanisms to prefetch memory locations in the cache. These must comply with the general cache behavior described in *Cache behavior* on page B2-5.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A3.9.4 キャッシュのプリロード

ARM アーキテクチャでは、メモリシステム ヒントとして PLD（データのプリロード）および PLI（命令のプリロード）が用意されており、ソフトウェアがどのメモリ位置の使用を予定しているかをハードウェアに伝達することができます。メモリシステムは、それらのヒントに対する応答として、そのメモリアクセスが発生する場合にアクセスを高速化するために必要なアクションを実行できます。これらのメモリシステム ヒントの作用は実装定義です。一般に、実装ではこの情報を使用して、データまたは命令の位置を通常のメモリより高速なアクセス速度を持つキャッシュに移動します。

プリロード命令はヒントなので、実装ではデバイスの機能的な動作に影響を与えない NOP として扱われます。この命令は同期データアボート例外を生成しませんが、例外的な状況下で、メモリシステムの動作が非同期アボートを生成することがあります。詳細については、*データアボート例外* P. B1-55 を参照して下さい。

ハードウェア実装が、キャッシュにメモリ位置をプリフェッチする実装固有の他の機構を提供することもあります。このような機構は、*キャッシュの動作* P. B2-5 で説明されている一般的なキャッシュ動作に準拠している必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter A4
The Instruction Sets

This chapter describes the ARM and Thumb instruction sets. It contains the following sections:

- *About the instruction sets* on page A4-2
- *Unified Assembler Language* on page A4-4
- *Branch instructions* on page A4-7
- *Data-processing instructions* on page A4-8
- *Status register access instructions* on page A4-18
- *Load/store instructions* on page A4-19
- *Load/store multiple instructions* on page A4-22
- *Miscellaneous instructions* on page A4-23
- *Exception-generating and exception-handling instructions* on page A4-24
- *Coprocessor instructions* on page A4-25
- *Advanced SIMD and VFP load/store instructions* on page A4-26
- *Advanced SIMD and VFP register transfer instructions* on page A4-29
- *Advanced SIMD data-processing operations* on page A4-30
- *VFP data-processing instructions* on page A4-38.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 A4 章
命令セット

本章では、ARM 命令セットと Thumb 命令セットについて説明します。本章は次のセクションから構成されています。

- *命令セットについて* P. A4-2
- *統一アセンブラ言語* P. A4-4
- *分岐命令* P. A4-7
- *データ処理命令* P. A4-8
- *ステータスレジスタ アクセス命令* P. A4-18
- *ロード / ストア命令* P. A4-19
- *複数ロード / ストア命令* P. A4-22
- *その他の命令* P. A4-23
- *例外生成命令と例外処理命令* P. A4-24
- *コプロセッサ命令* P. A4-25
- *アドバンスト SIMD および VFP のロード / ストア命令* P. A4-26
- *アドバンスト SIMD および VFP のレジスタ転送命令* P. A4-29
- *アドバンスト SIMD のデータ処理操作* P. A4-30
- *VFP のデータ処理命令* P. A4-38

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.1 About the instruction sets

ARMv7 contains two main instruction sets, the ARM and Thumb instruction sets. Much of the functionality available is identical in the two instruction sets. This chapter describes the functionality available in the instruction sets, and the *Unified Assembler Language* (UAL) that can be assembled to either instruction set.

The two instruction sets differ in how instructions are encoded:

- Thumb instructions are either 16-bit or 32-bit, and are aligned on a two-byte boundary. 16-bit and 32-bit instructions can be intermixed freely. Many common operations are most efficiently executed using 16-bit instructions. However:
  - — Most 16-bit instructions can only access eight of the general-purpose registers, R0-R7. These are known as the low registers. A small number of 16-bit instructions can access the high registers, R8-R15.
  - — Many operations that would require two or more 16-bit instructions can be more efficiently executed with a single 32-bit instruction.
- ARM instructions are always 32-bit, and are aligned on a four-byte boundary.

The ARM and Thumb instruction sets can *interwork* freely, that is, different procedures can be compiled or assembled to different instruction sets, and still be able to call each other efficiently.

ThumbEE is a variant of the Thumb instruction set that is designed as a target for dynamically generated code. However, it cannot interwork freely with the ARM and Thumb instruction sets.

See:
- Chapter A5 *ARM Instruction Set Encoding* for encoding details of the ARM instruction set
- Chapter A6 *Thumb Instruction Set Encoding* for encoding details of the Thumb instruction set
- Chapter A8 *Instruction Details* for detailed descriptions of the instructions
- Chapter A9 *ThumbEE* for encoding details of the ThumbEE instruction set.

A4.1.1 Changing between Thumb state and ARM state

A processor in Thumb state (that is, executing Thumb instructions) can enter ARM state (and change to executing ARM instructions) by executing any of the following instructions: `BX`, `BLX`, or an `LDR` or `LDM` that loads the PC.

A processor in ARM state (that is, executing ARM instructions) can enter Thumb state (and change to executing Thumb instructions) by executing any of the same instructions.

In ARMv7, a processor in ARM state can also enter Thumb state (and change to executing Thumb instructions) by executing an `ADC`, `ADD`, `AND`, `ASR`, `BIC`, `EOR`, `LSL`, `LSR`, `MOV`, `MVN`, `ORR`, `ROR`, `RRX`, `RSB`, `RSC`, `SBC`, or `SUB` instruction that has the PC as destination register and does not set the condition flags.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.1 命令セットについて

ARMv7 には、ARM 命令セットと Thumb 命令セットの 2 つの主要な命令セットが含まれます。2 つの命令セットの機能の大半は同じです。本章では、命令セットに備わっている機能と、どちらの命令セットにもアセンブルできる*統一アセンブラ言語*(UAL) について説明します。

2 つの命令セットは、命令のエンコード方法が異なっています。

- Thumb 命令は 16 ビットまたは 32 ビットで、2 バイト境界にアラインされています。16 ビットと 32 ビットの命令は自由に混在できます。多くの一般的な操作は、16 ビット命令を使用した場合に最も効率的に実行されます。ただし、以下のような制限があります。
  - – ほとんどの 16 ビット命令は、R0 ～ R7 の 8 つの汎用レジスタにしかアクセスできません。これらのレジスタを、下位レジスタと呼びます。一部の 16 ビット命令は、R8 ～ R15 の上位レジスタにアクセスできます。
  - – 2 つ以上の 16 ビット命令を必要とする多くの操作は、1 つの 32 ビット命令によってより効率的に実行できます。
- ARM 命令は常に 32 ビットで、4 バイト境界にアラインされています。

ARM 命令セットと Thumb 命令セットは自由に*インターワーキング*が可能です。つまり、各プロシージャを異なる命令セットにコンパイルまたはアセンブルしても、効率的に呼び出し合うことが可能です。

ThumbEE は、動的に生成されるコードのターゲットとして設計された Thumb 命令セットのバリアントです。しかし、これは ARM 命令セットや Thumb 命令セットとの自由なインターワーキングができません。

次の章を参照して下さい。
- 第 A5 章*「ARM 命令セットのエンコード」*（ARM 命令セットのエンコードの詳細）
- 第 A6 章*「Thumb 命令セットのエンコード」*（Thumb 命令セットのエンコードの詳細）
- 第 A8 章*「命令の詳細」*（命令の詳細な説明）
- 第 A9 章*「ThumbEE」*（ThumbEE 命令セットのエンコードの詳細）

A4.1.1 Thumb 状態と ARM 状態との切り替え

Thumb 状態の（つまり、Thumb 命令を実行している）プロセッサは、`BX`、`BLX`、または PC をロードする `LDR` か `LDM` のいずれかの命令を実行すると ARM 状態を開始（そして、ARM 命令の実行に変更）できます。

ARM 状態の（つまり、ARM 命令を実行している）プロセッサも、同じ命令のいずれかを実行すると Thumb 状態を開始（そして、Thumb 命令の実行に変更）できます。

ARMv7 では、ARM 状態のプロセッサは、PC をデスティネーションレジスタとし、条件フラグをセットしない `ADC`、`ADD`、`AND`、`ASR`、`BIC`、`EOR`、`LSL`、`LSR`、`MOV`、`MVN`、`ORR`、`ROR`、`RRX`、`RSB`、`RSC`、`SBC`、`SUB` のいずれかの命令を実行すると、Thumb 状態を開始（そして、Thumb 命令の実行に変更）できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

This permits calls and returns between ARM code written for ARMv4 processors and Thumb code running on ARMv7 processors to function correctly. In new code, ARM recommends that you use `BX` or `BLX` instructions instead. In particular, use `BX LR` to return from a procedure, not `MOV PC,LR`.

The target instruction set is either encoded directly in the instruction (for the immediate offset version of `BLX`), or is held as bit [0] of an *interworking address*. For details, see the description of the `BXWritePC()` function in *Pseudocode details of operations on ARM core registers* on page A2-12.

Exception entries and returns can also change between ARM and Thumb states. For details see *Exceptions* on page B1-30.

A4.1.2 Conditional execution

Most ARM instructions can be *conditionally executed*. This means that they only have their normal effect on the programmers' model operation, memory and coprocessors if the N, Z, C and V flags in the APSR satisfy a condition specified in the instruction. If the flags do not satisfy this condition, the instruction acts as a NOP, that is, execution advances to the next instruction as normal, including any relevant checks for exceptions being taken, but has no other effect.

Most Thumb instructions are unconditional. Conditional execution in Thumb code can be achieved using any of the following instructions:

- A 16-bit conditional branch instruction, with a branch range of –256 to +254 bytes. For details see *B* on page A8-44. Before ARMv6T2, this was the only mechanism for conditional execution in Thumb code.
- A 32-bit conditional branch instruction, with a branch range of approximately ± 1MB. For details see *B* on page A8-44.
- 16-bit Compare and Branch on Zero and Compare and Branch on Nonzero instructions, with a branch range of +4 to +130 bytes. For details see *CBNZ, CBZ* on page A8-66.
- A 16-bit If-Then instruction that makes up to four following instructions conditional. For details see *IT* on page A8-104. The instructions that are made conditional by an `IT` instruction are called its *IT block*. Instructions in an IT block must either all have the same condition, or some can have one condition, and others can have the inverse condition.

For more information about conditional execution see *Conditional execution* on page A8-8.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

これにより、ARMv4 プロセッサ用に作成されれた ARM コードと、ARMv7 プロセッサで実行される Thumb コードとの間で、呼び出しと復帰が正常に機能します。新しいコードでは、`BX` または `BLX` 命令の使用をお勧めします。特にプロシージャから復帰する際は、`MOV PC,LR` ではなく `BX LR` を使用して下さい。

ターゲットとする命令セットは、命令内に直接エンコードされるか（`BLX` のイミディエートオフセット バージョン）、*インターワーキングアドレス*のビット [0] に保持されます。詳細については、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 にある `BXWritePC()` 関数の説明を参照して下さい。

例外の開始と復帰でも、ARM 状態と Thumb 状態が切り替わることがあります。詳細については、*例外* P. B1-30 を参照して下さい。

A4.1.2 条件付き実行

ほとんどの ARM 命令は、*条件付き実行*が可能です。これは、APSR の N、Z、C、V フラグがその命令に指定された条件を満たす場合のみ、命令がプログラマモデルの動作、メモリ、コプロセッサに通常の効果を及ぼすことを意味します。フラグがこの条件を満たさない場合、命令は NOP として動作します。つまり、実行は次の命令へと通常に進み、例外処理をするかのチェックも行われますが、他の影響は何も及ぼしません。

ほとんどの Thumb 命令は無条件です。Thumb コードの条件付き実行は、次のいずれかの命令を使用して行われます。

- 16 ビットの条件付き分岐命令（分岐範囲 – 256 ～ + 254 バイト）。詳細については、*B* P. A8-44 を参照して下さい。ARMv6T2 より前は、これが Thumb コードで条件実行を行う唯一の手法でした。
- 32 ビットの条件付き分岐命令（分岐範囲約 ± 1MB）。詳細については、*B* P. A8-44 を参照して下さい。
- 16 ビットの比較して 0 で分岐、および比較して非 0 で分岐命令（分岐範囲 + 4 ～ + 130 バイト）。詳細については、*CBNZ*、*CBZ* P. A8-66 を参照して下さい。
- 次に続く最大 4 つの命令を条件付き実行とする、16 ビットの If-Then 命令。詳細については、*IT* P. A8-104 を参照して下さい。`IT` 命令によって条件付き実行に変更される命令を、*IT ブロック*と呼びます。IT ブロック内の命令はすべて同じ条件付きで実行するか、または一部の命令をある条件で、他の命令をその逆の条件で実行することができます。

条件付き実行の詳細については、*条件付き実行* P. A8-8 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.2 Unified Assembler Language

This document uses the ARM *Unified Assembler Language* (UAL). This assembly language syntax provides a canonical form for all ARM and Thumb instructions.

UAL describes the syntax for the mnemonic and the operands of each instruction. In addition, it assumes that instructions and data items can be given labels. It does not specify the syntax to be used for labels, nor what assembler directives and options are available. See your assembler documentation for these details.

Most earlier ARM assembly language mnemonics are still supported as synonyms, as described in the instruction details.

Note

Most earlier Thumb assembly language mnemonics are *not* supported. For details see Appendix C *Legacy Instruction Mnemonics*.

UAL includes *instruction selection* rules that specify which instruction encoding is selected when more than one can provide the required functionality. For example, both 16-bit and 32-bit encodings exist for an `ADD R0,R1,R2` instruction. The most common instruction selection rule is that when both a 16-bit encoding and a 32-bit encoding are available, the 16-bit encoding is selected, to optimize code density.

Syntax options exist to override the normal instruction selection rules and ensure that a particular encoding is selected. These are useful when disassembling code, to ensure that subsequent assembly produces the original code, and in some other situations.

A4.2.1 Conditional instructions

For maximum portability of UAL assembly language between the ARM and Thumb instruction sets, ARM recommends that:

- `IT` instructions are written before conditional instructions in the correct way for the Thumb instruction set.
- When assembling to the ARM instruction set, assemblers check that any `IT` instructions are correct, but do not generate any code for them.

Although other Thumb instructions are unconditional, all instructions that are made conditional by an `IT` instruction must be written with a condition. These conditions must match the conditions imposed by the `IT` instruction. For example, an `ITTEE EQ` instruction imposes the `EQ` condition on the first two following instructions, and the `NE` condition on the next two. Those four instructions must be written with `EQ`, `EQ`, `NE` and `NE` conditions respectively.

Some instructions cannot be made conditional by an `IT` instruction. Some instructions can be conditional if they are the last instruction in the IT block, but not otherwise.

The branch instruction encodings that include a condition field cannot be made conditional by an `IT` instruction. If the assembler syntax indicates a conditional branch that correctly matches a preceding `IT` instruction, it is assembled using a branch instruction encoding that does not include a condition field.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.2 統一アセンブラ言語

本書では、ARM *統一アセンブラ言語*(UAL) を使用します。このアセンブラ言語構文は、ARM および Thumb 命令の標準的な形式を示すものです。

UAL は、各命令のニーモニックとオペランドの構文を記述します。また、命令とデータの項目にはラベルを付けることができます。ラベルに使用される構文や、どのようなアセンブラディレクティブやオプションが使用できるかは指定されていません。これらの詳細については、アセンブラのドキュメントを参照して下さい。

以前の ARM アセンブラ言語ニーモニックのほとんどは、命令の詳細で示すように同義語としてサポートされています。

注

以前の Thumb アセンブラ言語ニーモニックのほとんどはサポート*されていません*。詳細については、付録 C*「過去の命令のニーモニック」*を参照して下さい。

UAL には*命令選択*規則があり、それにより、複数の命令エンコードが目的の機能を提供できる場合に、どのエンコードを選択するかが指定されます。例えば、`ADD R0,R1,R2` 命令には 16 ビットと 32 ビットの両方のエンコードが存在します。最も一般的な命令選択規則は、16 ビットエンコードと 32 ビットエンコードの両方が使用できる場合に、コード密度を最適化するために 16 ビットエンコードを選択するというものです。

通常の命令選択規則をオーバーライドし、特定のエンコードが選択されることを保証する構文オプションが存在します。コードの逆アセンブルを行うときに、生成されるアセンブリが元のコードを生成することを保証するようにするなどの、いくつかの他の状況においてこれらのオプションは有用です。

A4.2.1 条件付き命令

ARM 命令セットと Thumb 命令セットとの間で UAL アセンブラ言語の移植性を最大にするため、次のことをお勧めします。

- `IT` 命令は、Thumb 命令セットの正しい方法で、条件付き命令の前に記述する。
- ARM 命令セットへアセンブルするとき、アセンブラはすべての `IT` 命令が正しいことをチェックするが、それらのコードは生成しない。

他の Thumb 命令は無条件ですが、`IT` 命令によって条件付きに変更されたすべての命令は条件付きで記述する必要があります。これらの条件は、`IT` 命令によって示された条件と一致する必要があります。例えば、`ITTEE EQ` 命令はその後に続く最初の 2 つの命令について `EQ` 条件を指定し、その次にある 2 つの命令には `NE` 条件を指定します。これら 4 つの命令は、それぞれ `EQ`、`EQ`、`NE`、`NE` 条件付きで記述する必要があります。

一部の命令は、`IT` 命令で条件付きにすることはできません。一部の命令は、IT ブロックの最後の命令である場合は条件付きにできますが、それ以外の場合は条件付きにできません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.2.2 Use of labels in UAL instruction syntax

The UAL syntax for some instructions includes the label of an instruction or a literal data item that is at a fixed offset from the instruction being specified. The assembler must:

1. Calculate the PC or Align(PC,4) value of the instruction. The PC value of an instruction is its address plus 4 for a Thumb instruction, or plus 8 for an ARM instruction. The Align(PC,4) value of an instruction is its PC value ANDed with 0xFFFFFFFC to force it to be word-aligned. There is no difference between the PC and Align(PC,4) values for an ARM instruction, but there can be for a Thumb instruction.
2. Calculate the offset from the PC or Align(PC,4) value of the instruction to the address of the labelled instruction or literal data item.
3. Assemble a *PC-relative* encoding of the instruction, that is, one that reads its PC or Align(PC,4) value and adds the calculated offset to form the required address.

   Note

   For instructions that can encode a subtraction operation, if the instruction cannot encode the calculated offset but can encode minus the calculated offset, the instruction encoding specifies a subtraction of minus the calculated offset.

The syntax of the following instructions includes a label:

- B, BL, and BLX (immediate). The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a sign-extended immediate offset that is added to the PC value of the instruction to form the target address of the branch.
- CBNZ and CBZ. The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a zero-extended immediate offset that is added to the PC value of the instruction to form the target address of the branch. They do not support backward branches.
- LDC, LDC2, LDR, LDRB, LDRD, LDRH, LDRSB, LDRSH, PLD, PLDW, PLI, and VLDR. The normal assembler syntax of these load instructions can specify the label of a literal data item that is to be loaded. The encodings of these instructions specify a zero-extended immediate offset that is either added to or subtracted from the Align(PC,4) value of the instruction to form the address of the data item. A few such encodings perform a fixed addition or a fixed subtraction and must only be used when that operation is required, but most contain a bit that specifies whether the offset is to be added or subtracted.

  When the assembler calculates an offset of 0 for the normal syntax of these instructions, it must assemble an encoding that adds 0 to the Align(PC,4) value of the instruction. Encodings that subtract 0 from the Align(PC,4) value cannot be specified by the normal syntax.

  There is an alternative syntax for these instructions that specifies the addition or subtraction and the immediate offset explicitly. In this syntax, the label is replaced by [PC, #+/-<imm>], where:

  +/- Is + or omitted to specify that the immediate offset is to be added to the Align(PC,4) value, or - if it is to be subtracted.

  <imm> Is the immediate offset.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

条件フィールドを含む分岐命令エンコードは、IT 命令で条件付きにすることはできません。アセンブラ構文で、前の IT 命令と正しく一致する条件付き分岐が示されている場合、その命令は条件フィールドを含まない分岐命令エンコードを使用してアセンブルされます。

A4.2.2 UAL 命令構文でのラベルの使用

一部の命令の UAL 構文には、指定される命令から固定オフセットにある命令のラベル、またはリテラルデータ項目が含まれます。アセンブラは、次の動作を行う必要があります。

1. 命令の PC または Align(PC,4) の値を計算します。命令の PC の値は、Thumb 命令の場合は命令のアドレス + 4、ARM 命令の場合は + 8 です。命令の Align(PC,4) の値は、PC の値に 0xFFFFFFFC とのビット AND を適用し、ワードアラインドとした値です。ARM 命令については PC と Align(PC,4) の値は同じですが、Thumb 命令の場合は異なる可能性があります。
2. 命令の PC または Align(PC,4) の値から、ラベルの付いている命令またはリテラルデータ項目のアドレスへのオフセットを計算します。
3. 命令の *PC 相対*エンコードをアセンブルします。つまり、PC または Align(PC,4) の値を読み出し、計算されたオフセットを加算すると、目的のアドレスが得られるようにします。

   注

   減算をエンコードできる命令の場合、命令に計算されたオフセットをエンコードできないが、計算されたオフセットの負数をエンコードできる場合は、命令のエンコードには計算されたオフセットの負数の減算が指定されます。

次の命令の構文にはラベルが含まれます。

- B、BL、BLX（イミディエート）。これらの命令のアセンブラ構文では、常に分岐先の命令のラベルが指定されます。これらの命令のエンコードでは、符号拡張されたイミディエートオフセットが指定され、そのオフセットが命令の PC の値に追加されて分岐先アドレスが計算されます。
- CBNZ および CBZ。これらの命令のアセンブラ構文では、常に分岐先の命令のラベルが指定されます。これらの命令のエンコードでは、ゼロ拡張されたイミディエートオフセットが指定され、そのオフセットが命令の PC の値に追加されて分岐先アドレスが計算されます。これらの命令は後方分岐をサポートしていません。
- LDC、LDC2、LDR、LDRB、LDRD、LDRH、LDRSB、LDRSH、PLD、PLDW、PLI、VLDR。これらのロード命令の通常のアセンブラ構文では、ロードされるリテラルデータ項目のラベルを指定できます。これらの命令のエンコードでは、ゼロ拡張されたイミディエートオフセットが指定され、そのオフセットが命令の Align(PC,4) の値に加算または減算され、データ項目のアドレスが計算されます。このようなエンコードのいくつかは、オフセットの加算か減算のどちらかに固定されており、その操作が必要な場合のみ使用する必要がありますが、ほとんどのエンコードにはオフセットを加算するか減算するかを指定するビットが含まれています。

  アセンブラは、これらの命令について通常の構文のオフセットとして 0 を計算する場合、命令の Align(PC,4) の値に 0 を加算するエンコードをアセンブルする必要があります。Align(PC,4) の値から 0 を減算するエンコードは通常の構文では指定できません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

This alternative syntax makes it possible to assemble the encodings that subtract 0 from the `Align(PC,4)` value, and to disassemble them to a syntax that can be re-assembled correctly.

- `ADR`. The normal assembler syntax for this instruction can specify the label of an instruction or literal data item whose address is to be calculated. Its encoding specifies a zero-extended immediate offset that is either added to or subtracted from the `Align(PC,4)` value of the instruction to form the address of the data item, and some opcode bits that determine whether it is an addition or subtraction.

  When the assembler calculates an offset of 0 for the normal syntax of this instruction, it must assemble the encoding that adds 0 to the `Align(PC,4)` value of the instruction. The encoding that subtracts 0 from the `Align(PC,4)` value cannot be specified by the normal syntax.

  There is an alternative syntax for this instruction that specifies the addition or subtraction and the immediate value explicitly, by writing them as additions `ADD <Rd>,PC,#<imm>` or subtractions `SUB <Rd>,PC,#<imm>`. This alternative syntax makes it possible to assemble the encoding that subtracts 0 from the `Align(PC,4)` value, and to disassemble it to a syntax that can be re-assembled correctly.

Note

ARM recommends that where possible, you avoid using:

- the alternative syntax for the `ADR`, `LDC`, `LDC2`, `LDR`, `LDRB`, `LDRD`, `LDRH`, `LDRSB`, `LDRSH`, `PLD`, `PLI`, `PLDW`, and `VLDR` instructions
- the encodings of these instructions that subtract 0 from the `Align(PC,4)` value.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

これらの命令には、イミディエートオフセットを加算するか減算するかを明示的に指定する代替構文があります。この構文では、次に示すようにラベルを `[PC, #+/-<imm>]` に置き換えます。

`+/-` `Align(PC,4)` の値にイミディエートオフセットを加算することを指定する場合は、`+` を指定するか、符号を省略します。減算する場合は `-` を指定します。

`<imm>` イミディエートオフセットを指定します。

この代替構文では、`Align(PC,4)` の値から 0 を減算するエンコードをアセンブルすることが可能で、そのエンコードを逆アセンブルして、再度正しくアセンブル可能な構文に戻すこともできます。

- `ADR`。この命令の通常のアセンブラ構文では、命令またはリテラルデータ項目のラベルを指定し、アドレスを計算することができます。この命令のエンコードでは、ゼロ拡張されたイミディエートオフセットが指定され、そのオフセットが命令の `Align(PC,4)` の値に加算または減算され、データ項目のアドレスが計算されます。また、オペコードのいくつかのビットは、オフセットに対して加算と減算のどちらを行うかの指定に使用されます。

  アセンブラは、これらの命令について通常の構文のオフセットとして 0 を使用して計算する場合、命令の `Align(PC,4)` の値に 0 を追加するエンコードをアセンブルする必要があります。`Align(PC,4)` の値から 0 を減算するエンコードは通常の構文では指定できません。

  この命令には、イミディエート値を加算するか減算するかを明示的に指定する代替構文があります。この構文では、加算の場合 `ADD <Rd>,PC,#<imm>`、減算の場合 `SUB <Rd>,PC,#<imm>` と記述します。この代替構文では、`Align(PC,4)` の値から 0 を減算するエンコードをアセンブルすることが可能で、そのエンコードを逆アセンブルして、再度正しくアセンブル可能な構文に戻すこともできます。

注

次の構文の使用はできる限り避けることをお勧めします。

- `ADR`、`LDC`、`LDC2`、`LDR`、`LDRB`、`LDRD`、`LDRH`、`LDRSB`、`LDRSH`、`PLD`、`PLI`、`PLDW`、`VLDR` 命令の代替構文
- これらの命令について、`Align(PC,4)` の値から 0 を減算するエンコード

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.3 Branch instructions

Table A4-1 summarizes the branch instructions in the ARM and Thumb instruction sets. In addition to providing for changes in the flow of execution, some branch instructions can change instruction set.

Table A4-1 Branch instructions

Instruction	See	Range (Thumb)	Range (ARM)
Branch to target address	*B* on page A8-44	+/–16MB	+/–32MB
Compare and Branch on Nonzero, Compare and Branch on Zero	*CBNZ, CBZ* on page A8-66	0-126B	[a]
Call a subroutine Call a subroutine, change instruction set[b]	*BL, BLX (immediate)* on page A8-58	+/–16MB +/–16MB	+/–32MB +/–32MB
Call a subroutine, optionally change instruction set	*BLX (register)* on page A8-60	Any	Any
Branch to target address, change instruction set	*BX* on page A8-62	Any	Any
Change to Jazelle state	*BXJ* on page A8-64	-	-
Table Branch (byte offsets) Table Branch (halfword offsets)	*TBB, TBH* on page A8-446	0-510B 0-131070B	[a]

a. These instructions do not exist in the ARM instruction set.
b. The range is determined by the instruction set of the BLX instruction, not of the instruction it branches to.

Branches to loaded and calculated addresses can be performed by LDR, LDM and data-processing instructions. For details see *Load/store instructions* on page A4-19, *Load/store multiple instructions* on page A4-22, *Standard data-processing instructions* on page A4-8, and *Shift instructions* on page A4-10.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.3 分岐命令

ARM 命令セットおよび Thumb 命令セットの分岐命令の要約を表 A4-1 に示します。実行のフローを変更する他、一部の分岐命令は命令セットを変更できます。

表 A4-1 分岐命令

命令	参照先	範囲 (Thumb)	範囲 (ARM)
ターゲットアドレスへの分岐	*B* P. A8-44	+/− 16MB	+/− 32MB
比較して非 0 で分岐、比較して 0 で分岐	*CBNZ*、*CBZ* P. A8-66	0−126B	[a]
サブルーチン呼び出し サブルーチン呼び出し、命令セットの変更[b]	*BL*、*BLX（イミディエート）* P. A8-58	+/− 16MB +/− 16MB	+/− 32MB +/− 32MB
サブルーチン呼び出し、オプションとして命令セットの変更	*BLX（レジスタ）* P. A8-60	任意	任意
ターゲットアドレスへの分岐、命令セットの変更	*BX* P. A8-62	任意	任意
Jazelle 状態への変更	*BXJ* P. A8-64	−	−
テーブル分岐（バイトオフセット） テーブル分岐（ハーフワードオフセット）	*TBB*、*TBH* P. A8-446	0−510B 0−131070B	[a]

a. これらの命令は ARM 命令セットには存在しません。
b. 範囲は、分岐先の命令ではなく BLX 命令の命令セットによって決まります。

ロードされたアドレスおよび計算されたアドレスへの分岐は、LDR、LDM、およびデータ処理命令によって実行されます。詳細については、*ロード／ストア命令* P. A4-19、*複数ロード／ストア命令* P. A4-22、*標準データ処理命令* P. A4-8、*シフト命令* P. A4-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996−1998, 2000, 2004−2008 ARM Limited. All rights reserved.

A4.4 Data-processing instructions

Core data-processing instructions belong to one of the following groups:

- *Standard data-processing instructions*. These instructions perform basic data-processing operations, and share a common format with some variations.
- *Shift instructions* on page A4-10.
- *Saturating instructions* on page A4-13.
- *Packing and unpacking instructions* on page A4-14.
- *Miscellaneous data-processing instructions* on page A4-15.
- *Parallel addition and subtraction instructions* on page A4-16.
- *Divide instructions* on page A4-17.

For extension data-processing instructions, see *Advanced SIMD data-processing operations* on page A4-30 and *VFP data-processing instructions* on page A4-38.

A4.4.1 Standard data-processing instructions

These instructions generally have a destination register Rd, a first operand register Rn, and a second operand. The second operand can be another register Rm, or an immediate constant.

If the second operand is an immediate constant, it can be:

- Encoded directly in the instruction.
- A *modified immediate constant* that uses 12 bits of the instruction to encode a range of constants. Thumb and ARM instructions have slightly different ranges of modified immediate constants. For details see *Modified immediate constants in Thumb instructions* on page A6-17 and *Modified immediate constants in ARM instructions* on page A5-9.

If the second operand is another register, it can optionally be shifted in any of the following ways:

`LSL` Logical Shift Left by 1-31 bits.
`LSR` Logical Shift Right by 1-32 bits.
`ASR` Arithmetic Shift Right by 1-32 bits.
`ROR` Rotate Right by 1-31 bits.
`RRX` Rotate Right with Extend. For details see *Shift and rotate operations* on page A2-5.

In Thumb code, the amount to shift by is always a constant encoded in the instruction. In ARM code, the amount to shift by is either a constant encoded in the instruction, or the value of a register Rs.

For instructions other than `CMN`, `CMP`, `TEQ`, and `TST`, the result of the data-processing operation is placed in the destination register. In the ARM instruction set, the destination register can be the PC, causing the result to be treated as an address to branch to. In the Thumb instruction set, this is only permitted for some 16-bit forms of the `ADD` and `MOV` instructions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.4 データ処理命令

主要なデータ処理命令は、次のいずれかのグループに属します。

- *標準データ処理命令*。これらの命令は、基本的なデータ処理操作を実行し、いくつかのバリアントで共通のフォーマットを持ちます。
- *シフト命令* P. A4- 10
- *飽和命令* P. A4- 13
- *パッキングおよびアンパッキング命令* P. A4- 14
- *その他のデータ処理命令* P. A4- 15
- *並列加算および減算命令* P. A4- 16
- *除算命令* P. A4- 17

拡張データ処理命令については、*アドバンスト SIMD のデータ処理操作* P. A4- 30 と *VFP のデータ処理命令* P. A4- 38 を参照して下さい。

A4.4.1 標準データ処理命令

これらの命令には一般に、デスティネーションレジスタ Rd、最初のオペランドレジスタ Rn、2 番目のオペランドがあります。2 番目のオペランドは、別のレジスタ Rm またはイミディエート定数です。

2 番目のオペランドがイミディエート定数のときは、次のいずれかです。

- 命令に直接エンコードされている。
- 命令の 12 ビットを使って定数の範囲をエンコードしている *修飾イミディエート定数*。Thumb 命令と ARM 命令では、修飾イミディエート定数の範囲が多少異なります。詳細については、*Thumb 命令での修飾イミディエート定数* P. A6- 17 および *ARM 命令での修飾イミディエート定数* P. A5- 9 を参照して下さい。

2 番目のオペランドが別のレジスタの場合、オプションとして次のいずれかのシフトを適用できます。

`LSL` 1 ～ 31 ビットの論理左シフト
`LSR` 1 ～ 32 ビットの論理右シフト
`ASR` 1 ～ 32 ビットの算術右シフト
`ROR` 1 ～ 31 ビットの右ローテート
`RRX` 拡張付き右ローテート。詳細については、*シフトおよびローテート演算* P. A2- 5 を参照して下さい。

Thumb コードでは、シフトのビット数は常に定数で、命令にエンコードされます。ARM コードでは、シフトのビット数は命令にエンコードされている定数か、レジスタ Rs の値です。

`CMN`、`CMP`、`TEQ`、`TST` 以外の命令の場合、データ処理操作の結果はデスティネーションレジスタに配置されます。ARM 命令セットでは、デスティネーションレジスタが PC の場合があります。この場合、結果は分岐先のアドレスとして扱われます。Thumb 命令セットでは、`ADD` および `MOV` 命令の一部の 16 ビット形式にのみ、このオプションが許可されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

These instructions can optionally set the condition code flags, according to the result of the operation. If they do not set the flags, existing flag settings from a previous instruction are preserved.

Table A4-2 summarizes the main data-processing instructions in the Thumb and ARM instruction sets. Generally, each of these instructions is described in three sections in Chapter A8 *Instruction Details*, one section for each of the following:

- `INSTRUCTION` (immediate) where the second operand is a modified immediate constant.
- `INSTRUCTION` (register) where the second operand is a register, or a register shifted by a constant.
- `INSTRUCTION` (register-shifted register) where the second operand is a register shifted by a value obtained from another register. These are only available in the ARM instruction set.

Table A4-2 Standard data-processing instructions

Instruction	Mnemonic	Notes
Add with Carry	ADC	-
Add	ADD	Thumb instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
Form PC-relative Address	ADR	First operand is the PC. Second operand is an immediate constant. Thumb instruction set uses a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.
Bitwise AND	AND	-
Bitwise Bit Clear	BIC	-
Compare Negative	CMN	Sets flags. Like ADD but with no destination register.
Compare	CMP	Sets flags. Like SUB but with no destination register.
Bitwise Exclusive OR	EOR	-
Copy operand to destination	MOV	Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. For details see *Shift instructions* on page A4-10. The ARM and Thumb instruction sets permit use of a modified immediate constant or a zero-extended 16-bit immediate constant.
Bitwise NOT	MVN	Has only one operand, with the same options as the second operand in most of these instructions.
Bitwise OR NOT	ORN	Not available in the ARM instruction set.
Bitwise OR	ORR	-

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

これらの命令は、オプションとして、演算の結果に基づいて条件コードフラグをセットできます。フラグをセットしない場合、前の命令でセットされたそれまでのフラグ状態が保持されます。

Thumb 命令セットおよび ARM 命令セットの主要なデータ処理命令の要約を表 A4-2 に示します。一般に、これらの各命令は、第 A8 章*「命令の詳細」*で 3 つのセクションに分けて説明され、各セクションでは次の場合について説明されています。

- `INSTRUCTION`（イミディエート）: 2 番目のオペランドが修飾されたイミディエート定数の場合
- `INSTRUCTION`（レジスタ）: 2 番目のオペランドがレジスタ、または定数によりシフトされたレジスタの場合
- `INSTRUCTION`（レジスタシフト レジスタ）: 2 番目のオペランドが、別のレジスタから取得した値のビット数だけシフトされたレジスタの場合。この形式は ARM 命令セットにのみ存在します。

表 A4-2 標準データ処理命令

命令	ニーモニック	注
キャリー付き加算	ADC	–
加算	ADD	Thumb 命令セットでは、修飾されたイミディエート定数、またはゼロ拡張された 12 ビットイミディエート定数が使用できます。
PC 相対アドレスの生成	ADR	最初のオペランドは PC です。2 番目のオペランドはイミディエート定数です。Thumb 命令セットでは、ゼロ拡張された 12 ビットイミディエート定数が使用されます。演算は加算または減算です。
ビット単位論理積	AND	–
ビット単位のビットクリア	BIC	–
比較否定	CMN	フラグをセットします。ADD と似ていますが、デスティネーションレジスタはありません。
比較	CMP	フラグをセットします。SUB と似ていますが、デスティネーションレジスタはありません。
ビット単位排他的論理和	EOR	–
オペランドをデスティネーションへコピー	MOV	オペランドは 1 つだけで、これらの命令のほとんどには 2 番目のオペランドと同じオプションがあります。オペランドがシフトされたレジスタの場合、この命令は実質的に LSL、LSR、ASR、ROR 命令のいずれかになります。詳細については、*シフト命令* P. A4-10 を参照して下さい。 ARM 命令セットおよび Thumb 命令セットでは、修飾されたイミディエート定数、またはゼロ拡張された 16 ビットイミディエート定数が使用できます。
ビット単位の NOT	MVN	オペランドは 1 つだけで、これらの命令のほとんどには 2 番目のオペランドと同じオプションがあります。
ビット単位否定論理和	ORN	ARM 命令セットには存在しません。
ビット単位論理和	ORR	–

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A4-2 Standard data-processing instructions (continued)

Instruction	Mnemonic	Notes
Reverse Subtract	RSB	Subtracts first operand from second operand. This permits subtraction from constants and shifted registers.
Reverse Subtract with Carry	RSC	Not available in the Thumb instruction set.
Subtract with Carry	SBC	-
Subtract	SUB	Thumb instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
Test Equivalence	TEQ	Sets flags. Like EOR but with no destination register.
Test	TST	Sets flags. Like AND but with no destination register.

A4.4.2 Shift instructions

Table A4-3 lists the shift instructions in the ARM and Thumb instruction sets.

Table A4-3 Shift instructions

Instruction	See
Arithmetic Shift Right	*ASR (immediate)* on page A8-40
Arithmetic Shift Right	*ASR (register)* on page A8-42
Logical Shift Left	*LSL (immediate)* on page A8-178
Logical Shift Left	*LSL (register)* on page A8-180
Logical Shift Right	*LSR (immediate)* on page A8-182
Logical Shift Right	*LSR (register)* on page A8-184
Rotate Right	*ROR (immediate)* on page A8-278
Rotate Right	*ROR (register)* on page A8-280
Rotate Right with Extend	*RRX* on page A8-282

In the ARM instruction set only, the destination register of these instructions can be the PC, causing the result to be treated as an address to branch to.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A4-2 標準データ処理命令（続き）

命令	ニーモニック	注
逆減算	RSB	2 番目のオペランドから最初のオペランドを減算します。これによって、定数やシフトされたレジスタからの減算を実行できます。
キャリー付き逆減算	RSC	Thumb 命令セットには存在しません。
キャリー付き減算	SBC	–
減算	SUB	Thumb 命令セットでは、修飾されたイミディエート定数、またはゼロ拡張された 12 ビットイミディエート定数が使用できます。
等価テスト	TEQ	フラグをセットします。EOR と似ていますが、デスティネーションレジスタはありません。
テスト	TST	フラグをセットします。AND と似ていますが、デスティネーションレジスタはありません。

A4.4.2 シフト命令

ARM 命令セットおよび Thumb 命令セットのシフト命令の一覧を、表 A4-3 に示します。

表 A4-3 シフト命令

命令	参照先
算術右シフト	*ASR（イミディエート）* P. A8-40
算術右シフト	*ASR（レジスタ）* P. A8-42
論理左シフト	*LSL（イミディエート）* P. A8-178
論理左シフト	*LSL（レジスタ）* P. A8-180
論理右シフト	*LSR（イミディエート）* P. A8-182
論理右シフト	*LSR（レジスタ）* P. A8-184
右ローテート	*ROR（イミディエート）* P. A8-278
右ローテート	*ROR（レジスタ）* P. A8-280
拡張付き右ローテート	*RRX* P. A8-282

ARM 命令セットでのみ、これらの命令のデスティネーションレジスタに PC を指定できます。この場合、結果は分岐先のアドレスとして扱われます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.4.3 Multiply instructions

These instructions can operate on signed or unsigned quantities. In some types of operation, the results are same whether the operands are signed or unsigned.

- Table A4-4 summarizes the multiply instructions where there is no distinction between signed and unsigned quantities.

  The least significant 32 bits of the result are used. More significant bits are discarded.
- Table A4-5 summarizes the signed multiply instructions.
- Table A4-6 on page A4-12 summarizes the unsigned multiply instructions.

Table A4-4 General multiply instructions

Instruction	See	Operation (number of bits)
Multiply Accumulate	*MLA* on page A8-190	32 = 32 + 32 x 32
Multiply and Subtract	*MLS* on page A8-192	32 = 32 – 32 x 32
Multiply	*MUL* on page A8-212	32 = 32 x 32

Table A4-5 Signed multiply instructions

Instruction	See	Operation (number of bits)
Signed Multiply Accumulate (halfwords)	*SMLABB, SMLABT, SMLATB, SMLATT* on page A8-330	32 = 32 + 16 x 16
Signed Multiply Accumulate Dual	*SMLAD* on page A8-332	32 = 32 + 16 x 16 + 16 x 16
Signed Multiply Accumulate Long	*SMLAL* on page A8-334	64 = 64 + 32 x 32
Signed Multiply Accumulate Long (halfwords)	*SMLALBB, SMLALBT, SMLALTB, SMLALTT* on page A8-336	64 = 64 + 16 x 16
Signed Multiply Accumulate Long Dual	*SMLALD* on page A8-338	64 = 64 + 16 x 16 + 16 x 16
Signed Multiply Accumulate (word by halfword)	*SMLAWB, SMLAWT* on page A8-340	32 = 32 + 32 x 16 [a]
Signed Multiply Subtract Dual	*SMLSD* on page A8-342	32 = 32 + 16 x 16 – 16 x 16
Signed Multiply Subtract Long Dual	*SMLSLD* on page A8-344	64 = 64 + 16 x 16 – 16 x 16
Signed Most Significant Word Multiply Accumulate	*SMMLA* on page A8-346	32 = 32 + 32 x 32 [b]

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.4.3 乗算命令

これらの命令は、符号付きまたは符号なしの数値に対して演算を行います。一部のタイプの演算では、オペランドが符号付き、符号なしのいずれでも結果は同じです。

- 符号付きと符号なしの数値で差のない乗算命令の要約を、表 A4-4 に示します。

  結果の下位 32 ビットが使用されます。上位ビットは破棄されます。
- 符号付き乗算命令の要約を、表 A4-5 に示します。
- 符号なし乗算命令の要約を、P. A4-12 の表 A4-6 に示します。

表 A4-4 汎用乗算命令

命令	参照先	動作（ビットの数）
積和演算	*MLA* P. A8-190	32 = 32 + 32 x 32
積減算	*MLS* P. A8-192	32 = 32 - 32 x 32
乗算	*MUL* P. A8-212	32 = 32 x 32

表 A4-5 符号付き乗算命令

命令	参照先	動作（ビットの数）
符号付き積和演算（ハーフワード）	*SMLABB*、*SMLABT*、*SMLATB*、*SMLATT* P. A8-330	32 = 32 + 16 x 16
符号付きデュアル積和演算	*SMLAD* P. A8-332	32 = 32 + 16 x 16 + 16 x 16
符号付きロング積和演算	*SMLAL* P. A8-334	64 = 64 + 32 x 32
符号付きロング積和演算（ハーフワード）	*SMLALBB*、*SMLALBT*、*SMLALTB*、*SMLALTT* P. A8-336	64 = 64 + 16 x 16
符号付きデュアルロング積和演算	*SMLALD* P. A8-338	64 = 64 + 16 x 16 + 16 x 16
符号付き積和演算（ワード × ハーフワード）	*SMLAWB*、*SMLAWT* P. A8-340	32 = 32 + 32 x 16 [a]
符号付きデュアル積減算	*SMLSD* P. A8-342	32 = 32 + 16 x 16 - 16 x 16
符号付きデュアルロング積減算	*SMLSLD* P. A8-344	64 = 64 + 16 x 16 - 16 x 16
符号付き最上位ワード積和演算	*SMMLA* P. A8-346	32 = 32 + 32 x 32 [b]
符号付き最上位ワード積減算	*SMMLS* P. A8-348	32 = 32 - 32 x 32 [b]

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A4-5 Signed multiply instructions (continued)

Instruction	See	Operation (number of bits)
Signed Most Significant Word Multiply Subtract	*SMMLS* on page A8-348	32 = 32 – 32 x 32 [b]
Signed Most Significant Word Multiply	*SMMUL* on page A8-350	32 = 32 x 32 [b]
Signed Dual Multiply Add	*SMUAD* on page A8-352	32 = 16 x 16 + 16 x 16
Signed Multiply (halfwords)	*SMULBB*, *SMULBT*, *SMULTB*, *SMULTT* on page A8-354	32 = 16 x 16
Signed Multiply Long	*SMULL* on page A8-356	64 = 32 x 32
Signed Multiply (word by halfword)	*SMULWB*, *SMULWT* on page A8-358	32 = 32 x 16 [a]
Signed Dual Multiply Subtract	*SMUSD* on page A8-360	32 = 16 x 16 – 16 x 16

a. The most significant 32 bits of the 48-bit product are used. Less significant bits are discarded.
b. The most significant 32 bits of the 64-bit product are used. Less significant bits are discarded.

Table A4-6 Unsigned multiply instructions

Instruction	See	Operation (number of bits)
Unsigned Multiply Accumulate Accumulate Long	*UMAAL* on page A8-482	64 = 32 + 32 + 32 x 32
Unsigned Multiply Accumulate Long	*UMLAL* on page A8-484	64 = 64 + 32 x 32
Unsigned Multiply Long	*UMULL* on page A8-486	64 = 32 x 32

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A4-5 符号付き乗算命令（続き）

命令	参照先	動作（ビットの数）
符号付き最上位ワード乗算	*SMMUL* P. A8-350	32 = 32 x 32 [b]
符号付きデュアル積加算	*SMUAD* P. A8-352	32 = 16 x 16 + 16 x 16
符号付き乗算（ハーフワード）	*SMULBB*、*SMULBT*、*SMULTB*、*SMULTT* P. A8-354	32 = 16 x 16
符号付きロング乗算	*SMULL* P. A8-356	64 = 32 x 32
符号付き乗算（ワード × ハーフワード）	*SMULWB*、*SMULWT* P. A8-358	32 = 32 x 16 [a]
符号付きデュアル積減算	*SMUSD* P. A8-360	32 = 16 x 16 - 16 x 16

a. 48 ビットの積の最上位 32 ビットが使用されます。下位ビットは破棄されます。
b. 64 ビットの積の最上位 32 ビットが使用されます。下位ビットは破棄されます。

表 A4-6 符号なし乗算命令

命令	参照先	動作（ビットの数）
符号なしロング積和和演算	*UMAAL* P. A8-482	64 = 32 + 32 + 32 x 32
符号なしロング積和演算	*UMLAL* P. A8-484	64 = 64 + 32 x 32
符号なしロング乗算	*UMULL* P. A8-486	64 = 32 x 32

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.4.4 Saturating instructions

Table A4-7 lists the saturating instructions in the ARM and Thumb instruction sets. For more information, see *Pseudocode details of saturation* on page A2-9.

Table A4-7 Saturating instructions

Instruction	See	Operation
Signed Saturate	*SSAT* on page A8-362	Saturates optionally shifted 32-bit value to selected range
Signed Saturate 16	*SSAT16* on page A8-364	Saturates two 16-bit values to selected range
Unsigned Saturate	*USAT* on page A8-504	Saturates optionally shifted 32-bit value to selected range
Unsigned Saturate 16	*USAT16* on page A8-506	Saturates two 16-bit values to selected range

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.4.4 飽和命令

ARM 命令セットおよび Thumb 命令セットの飽和命令の一覧を、表 A4-7 に示します。詳細については、*飽和の擬似コードの詳細* P. A2-9 を参照して下さい。

表 A4-7 飽和命令

命令	参照先	動作
符号付き飽和	*SSAT* P. A8-362	32 ビットの値を選択された範囲に飽和します。オプションとして、値をシフトすることもできます。
符号付き飽和 16	*SSAT16* P. A8-364	2 つの 16 ビットの値を選択された範囲に飽和します。
符号なし飽和	*USAT* P. A8-504	32 ビットの値を選択された範囲に飽和します。オプションとして、値をシフトすることもできます。
符号なし飽和 16	*USAT16* P. A8-506	2 つの 16 ビットの値を選択された範囲に飽和します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.4.5 Packing and unpacking instructions

Table A4-8 lists the packing and unpacking instructions in the ARM and Thumb instruction sets. These are all available from ARMv6T2 in the Thumb instruction set, and from ARMv6 onwards in the ARM instruction set.

Table A4-8 Packing and unpacking instructions

Instruction	See	Operation
Pack Halfword	*PKH* on page A8-234	Combine halfwords
Signed Extend and Add Byte	*SXTAB* on page A8-434	Extend 8 bits to 32 and add
Signed Extend and Add Byte 16	*SXTAB16* on page A8-436	Dual extend 8 bits to 16 and add
Signed Extend and Add Halfword	*SXTAH* on page A8-438	Extend 16 bits to 32 and add
Signed Extend Byte	*SXTB* on page A8-440	Extend 8 bits to 32
Signed Extend Byte 16	*SXTB16* on page A8-442	Dual extend 8 bits to 16
Signed Extend Halfword	*SXTH* on page A8-444	Extend 16 bits to 32
Unsigned Extend and Add Byte	*UXTAB* on page A8-514	Extend 8 bits to 32 and add
Unsigned Extend and Add Byte 16	*UXTAB16* on page A8-516	Dual extend 8 bits to 16 and add
Unsigned Extend and Add Halfword	*UXTAH* on page A8-518	Extend 16 bits to 32 and add
Unsigned Extend Byte	*UXTB* on page A8-520	Extend 8 bits to 32
Unsigned Extend Byte 16	*UXTB16* on page A8-522	Dual extend 8 bits to 16
Unsigned Extend Halfword	*UXTH* on page A8-524	Extend 16 bits to 32

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.4.5 パッキングおよびアンパッキング命令

ARM 命令セットおよび Thumb 命令セットのパッキングおよびアンパッキング命令の一覧を、表 A4-8 に示します。これらはすべて、ARMv6T2 またはそれ以降の Thumb 命令セット、および ARMv6 またはそれ以降の ARM 命令セットに存在します。

表 A4-8 パッキングおよびアンパッキング命令

命令	参照先	動作
パックハーフワード	*PKH* P. A8-234	ハーフワードを組み合わせます。
符号付きバイト拡張および加算	*SXTAB* P. A8-434	8 ビットを 32 ビットに拡張して加算します。
符号付きバイト拡張および加算 16	*SXTAB16* P. A8-436	8 ビットを 16 ビットにデュアル拡張して加算します。
符号付きハーフワード拡張および加算	*SXTAH* P. A8-438	16 ビットを 32 ビットに拡張して加算します。
符号付きバイト拡張	*SXTB* P. A8-440	8 ビットを 32 ビットに拡張します。
符号付きバイト拡張 16	*SXTB16* P. A8-442	8 ビットを 16 ビットにデュアル拡張します。
符号付きハーフワード拡張	*SXTH* P. A8-444	16 ビットを 32 ビットに拡張します。
符号なしバイト拡張および加算	*UXTAB* P. A8-514	8 ビットを 32 ビットに拡張して加算します。
符号なしバイト拡張および加算 16	*UXTAB16* P. A8-516	8 ビットを 16 ビットにデュアル拡張して加算します。
符号なしハーフワード拡張および加算	*UXTAH* P. A8-518	16 ビットを 32 ビットに拡張して加算します。
符号なしバイト拡張	*UXTB* P. A8-520	8 ビットを 32 ビットに拡張します。
符号なしバイト拡張 16	*UXTB16* P. A8-522	8 ビットを 16 ビットにデュアル拡張します。
符号なしハーフワード拡張	*UXTH* P. A8-524	16 ビットを 32 ビットに拡張します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.4.6 Miscellaneous data-processing instructions

Table A4-9 lists the miscellaneous data-processing instructions in the ARM and Thumb instruction sets. Immediate values in these instructions are simple binary numbers.

Table A4-9 Miscellaneous data-processing instructions

Instruction	See	Notes
Bit Field Clear	*BFC* on page A8-46	-
Bit Field Insert	*BFI* on page A8-48	-
Count Leading Zeros	*CLZ* on page A8-72	-
Move Top	*MOVT* on page A8-200	Moves 16-bit immediate value to top halfword. Bottom halfword unchanged.
Reverse Bits	*RBIT* on page A8-270	-
Byte-Reverse Word	*REV* on page A8-272	-
Byte-Reverse Packed Halfword	*REV16* on page A8-274	-
Byte-Reverse Signed Halfword	*REVSH* on page A8-276	-
Signed Bit Field Extract	*SBFX* on page A8-308	-
Select Bytes using GE flags	*SEL* on page A8-312	-
Unsigned Bit Field Extract	*UBFX* on page A8-466	-
Unsigned Sum of Absolute Differences	*USAD8* on page A8-500	-
Unsigned Sum of Absolute Differences and Accumulate	*USADA8* on page A8-502	-

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.4.6 その他のデータ処理命令

ARM 命令セットおよび Thumb 命令セットに含まれるその他のデータ処理命令の一覧を、表 A4-9 に示します。これらの命令に含まれるイミディエート値は、単純なバイナリ数値です。

表 A4-9 その他のデータ処理命令

命令	参照先	注
ビッドフィールドのクリア	*BFC* P. A8-46	–
ビットフィールドの挿入	*BFI* P. A8-48	–
先行ゼロカウント	*CLZ* P. A8-72	–
上位ビットへの移動	*MOVT* P. A8-200	16 ビットのイミディエート値を上位ハーフワードへ移動します。下位ハーフワードは変更されません。
ビット反転	*RBIT* P. A8-270	–
ワードのバイト反転	*REV* P. A8-272	–
パックハーフワードのバイト反転	*REV16* P. A8-274	–
符号付きハーフワードのバイト反転	*REVSH* P. A8-276	–
符号付きビットフィールドの抽出	*SBFX* P. A8-308	–
GE フラグを使用してバイトを選択	*SEL* P. A8-312	–
符号なしビットフィールドの抽出	*UBFX* P. A8-466	–
符号なし絶対差の和	*USAD8* P. A8-500	–
符号なし絶対差の和および累積	*USADA8* P. A8-502	–

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.4.7 Parallel addition and subtraction instructions

These instructions perform additions and subtractions on the values of two registers and write the result to a destination register, treating the register values as sets of two halfwords or four bytes. They are available in ARMv6 and above.

These instructions consist of a prefix followed by a main instruction mnemonic. The prefixes are as follows:

`S` Signed arithmetic modulo 2^8 or 2^{16}.

`Q` Signed saturating arithmetic.

`SH` Signed arithmetic, halving the results.

`U` Unsigned arithmetic modulo 2^8 or 2^{16}.

`UQ` Unsigned saturating arithmetic.

`UH` Unsigned arithmetic, halving the results.

The main instruction mnemonics are as follows:

`ADD16` Adds the top halfwords of two operands to form the top halfword of the result, and the bottom halfwords of the same two operands to form the bottom halfword of the result.

`ASX` Exchanges halfwords of the second operand, and then adds top halfwords and subtracts bottom halfwords.

`SAX` Exchanges halfwords of the second operand, and then subtracts top halfwords and adds bottom halfwords.

`SUB16` Subtracts each halfword of the second operand from the corresponding halfword of the first operand to form the corresponding halfword of the result.

`ADD8` Adds each byte of the second operand to the corresponding byte of the first operand to form the corresponding byte of the result.

`SUB8` Subtracts each byte of the second operand from the corresponding byte of the first operand to form the corresponding byte of the result.

The instruction set permits all 36 combinations of prefix and main instruction operand.

See also *Advanced SIMD parallel addition and subtraction* on page A4-31.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.4.7 並列加算および減算命令

これらの命令は、2 つのレジスタ値に対して加算と減算を実行し、結果をデスティネーションレジスタに書き込みます。レジスタの値は、2 つのハーフワードまたは 4 つのバイトのセットとして扱われます。ARMv6 およびそれ以降で使用できる命令です。

これらの命令では、中心となる命令ニーモニックの前に接頭文字があります。接頭文字は次に示すとおりです。

`S` 符号付きのモジュロ 2^8 または 2^{16} 算術演算

`Q` 符号付き飽和算術演算

`SH` 結果を半減する符号付き算術演算

`U` 符号なしのモジュロ 2^8 または 2^{16} 算術演算

`UQ` 符号なし飽和算術演算

`UH` 結果を半減する符号なし算術演算

中心となる命令ニーモニックは次に示すとおりです。

`ADD16` 2 つのオペランドの最上位ハーフワードを加算し、結果の最上位ハーフワードとします。同じ 2 つのオペランドの最下位ハーフワードも加算し、結果の最下位ハーフワードとします。

`ASX` 2 番目のオペランドのハーフワードを交換してから、最上位のハーフワードを加算、最下位のハーフワードを減算します。

`SAX` 2 番目のオペランドのハーフワードを交換してから、最上位のハーフワードを減算、最下位のハーフワードを加算します。

`SUB16` 2 番目のオペランドの各ハーフワードを、最初のオペランドの対応するハーフワードから減算し、結果の対応するハーフワードとします。

`ADD8` 2 番目のオペランドの各バイトを、最初のオペランドの対応するバイトに加算し、結果の対応するバイトとします。

`SUB8` 2 番目のオペランドの各バイトを、最初のオペランドの対応するバイトから減算し、結果の対応するバイトとします。

命令セットでは、接頭文字と中心となる命令オペランドについて、可能な 36 通りの組み合わせすべてが使用できます。

アドバンスト SIMD の並列加算および減算 P. A4-31 も参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.4.8 Divide instructions

In the ARMv7-R profile, the Thumb instruction set includes signed and unsigned integer divide instructions that are implemented in hardware. For details of the instructions see:

- *SDIV* on page A8-310
- *UDIV* on page A8-468.

Note

- SDIV and UDIV are UNDEFINED in the ARMv7-A profile.
- The ARMv7-M profile also includes the SDIV and UDIV instructions.

In the ARMv7-R profile, the SCTLR.DZ bit enables divide by zero fault detection, see *c1, System Control Register (SCTLR)* on page B4-45:

DZ == 0 Divide-by-zero returns a zero result.

DZ == 1 SDIV and UDIV generate an Undefined Instruction exception on a divide-by-zero.

The SCTLR.DZ bit is cleared to zero on reset.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.4.8 除算命令

ARMv7-R プロファイルでは、Thumb 命令セットに符号付きおよび符号なしの整数除算命令が含まれており、これらの命令はハードウェアで実装されます。命令の詳細については、次の項目を参照して下さい。

- *SDIV* P. A8-310
- *UDIV* P. A8-468

注

- SDIV および UDIV 命令は、ARMv7-A プロファイルで未定義です。
- ARMv7-M プロファイルには、SDIV および UDIV 命令も含まれます。

ARMv7-R プロファイルでは、SCTLR.DZ ビットによって、0 による除算のフォールトの検出が可能になります。*c1、システム制御レジスタ (SCTLR)* P. B4-45 を参照して下さい。

DZ = 0 0 による除算を行った場合、結果として 0 が返されます。

DZ = 1 SDIV および UDIV で 0 による除算を行った場合、未定義命令例外が生成されます。

SCTLR.DZ ビットはリセット時に 0 にクリアされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.5 Status register access instructions

The MRS and MSR instructions move the contents of the *Application Program Status Register* (APSR) to or from a general-purpose register.

The APSR is described in *The Application Program Status Register (APSR)* on page A2-14.

The condition flags in the APSR are normally set by executing data-processing instructions, and are normally used to control the execution of conditional instructions. However, you can set the flags explicitly using the MSR instruction, and you can read the current state of the flags explicitly using the MRS instruction.

For details of the system level use of status register access instructions CPS, MRS, and MSR, see Chapter B6 *System Instructions*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.5 ステータスレジスタ アクセス命令

MRS および MSR 命令は、*アプリケーションプログラム ステータスレジスタ*(APSR) の内容を汎用レジスタへ、またはその逆に移動します。

APSR については、*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2- 14 を参照して下さい。

APSR の条件フラグは、一般にデータ処理命令の実行によりセットされ、条件付き命令の実行の制御に使用されます。ただし、MSR 命令を使用して明示的にフラグをセットでき、MRS 命令を使用して明示的に現在のフラグの状態を読み出すことができます。

ステータスレジスタ アクセス命令 CPS、MRS、MSR をシステムレベルで使用する方法の詳細については、第 B6 章*「システム命令」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.6 Load/store instructions

Table A4-10 summarizes the general-purpose register load/store instructions in the ARM and Thumb instruction sets. See also:

- *Load/store multiple instructions* on page A4-22
- *Advanced SIMD and VFP load/store instructions* on page A4-26.

Load/store instructions have several options for addressing memory. For more information, see *Addressing modes* on page A4-20.

Table A4-10 Load/store instructions

Data type	Load	Store	Load unprivileged	Store unprivileged	Load-Exclusive	Store-Exclusive
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX
16-bit halfword	-	STRH	-	STRHT	-	STREXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-
Two 32-bit words	LDRD	STRD	-	-	-	-
64-bit doubleword	-	-	-	-	LDREXD	STREXD

A4.6.1 Loads to the PC

The LDR instruction can be used to load a value into the PC. The value loaded is treated as an interworking address, as described by the LoadWritePC() pseudocode function in *Pseudocode details of operations on ARM core registers* on page A2-12.

A4.6.2 Halfword and byte loads and stores

Halfword and byte stores store the least significant halfword or byte from the register, to 16 or 8 bits of memory respectively. There is no distinction between signed and unsigned stores.

Halfword and byte loads load 16 or 8 bits from memory into the least significant halfword or byte of a register. Unsigned loads zero-extend the loaded value to 32 bits, and signed loads sign-extend the value to 32 bits.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.6 ロード / ストア命令

ARM 命令セットおよび Thumb 命令セットに含まれる汎用レジスタのロード / ストア命令の要約を、表 A4-10 に示します。次に示すセクションも参照して下さい。

- *複数ロード / ストア命令* P. A4-22
- *アドバンスト SIMD および VFP のロード / ストア命令* P. A4-26

ロード / ストア命令には、メモリのアドレスを指定するいくつかのオプションがあります。詳細については、*アドレシングモード* P. A4-20 を参照して下さい。

表 A4-10 ロード / ストア命令

データタイプ	ロード	ストア	非特権ロード	非特権ストア	排他ロード	排他ストア
32 ビットワード	LDR	STR	LDRT	STRT	LDREX	STREX
16 ビットハーフワード	–	STRH	–	STRHT	–	STREXH
16 ビット符号なしハーフワード	LDRH	–	LDRHT	-	LDREXH	–
16 ビット符号付きハーフワード	LDRSH	–	LDRSHT	–	–	–
8 ビットバイト	–	STRB	–	STRBT	–	STREXB
8 ビット符号なしバイト	LDRB	-	LDRBT	–	LDREXB	–
8 ビット符号付きバイト	LDRSB	–	LDRSBT	–	–	–
2 つの 32 ビットワード	LDRD	STRD	–	–	–	–
64 ビットのダブルワード	–	–	–	–	LDREXD	STREXD

A4.6.1 PC へのロード

LDR 命令は、値を PC へロードするために使用できます。ロードされた値は、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 の LoadWritePC() 擬似コード関数で説明されているように、インターワーキングアドレスとみなされます。

A4.6.2 ハーフワードとバイトのロード / ストア

ハーフワードとバイトのストアは、レジスタの下位ハーフワードまたはバイトを、それぞれ 16 または 8 ビットのメモリへストアします。ストアには符号付きと符号なしの区別はありません。

ハーフワードとバイトのロードは、16 または 8 ビットのメモリから、レジスタの下位ハーフワードまたはバイトへロードします。符号なしロードの場合はロードされた値が 32 ビットにゼロ拡張され、符号付きロードの場合は値が 32 ビットに符号拡張されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.6.3 Unprivileged loads and stores

In an unprivileged mode, unprivileged loads and stores operate in exactly the same way as the corresponding ordinary operations. In a privileged mode, unprivileged loads and stores are treated as though they were executed in an unprivileged mode. For more information, see *Privilege level access controls for data accesses* on page A3-38.

A4.6.4 Exclusive loads and stores

Exclusive loads and stores provide for shared memory synchronization. For more information, see *Synchronization and semaphores* on page A3-12.

A4.6.5 Addressing modes

The address for a load or store is formed from two parts: a value from a base register, and an offset.

The base register can be any one of the general-purpose registers.

For loads, the base register can be the PC. This permits PC-relative addressing for position-independent code. Instructions marked (literal) in their title in Chapter A8 *Instruction Details* are PC-relative loads.

The offset takes one of three formats:

Immediate The offset is an unsigned number that can be added to or subtracted from the base register value. Immediate offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets and input/output registers.

Register The offset is a value from a general-purpose register. This register cannot be the PC. The value can be added to, or subtracted from, the base register value. Register offsets are useful for accessing arrays or blocks of data.

Scaled register The offset is a general-purpose register, other than the PC, shifted by an immediate value, then added to or subtracted from the base register. This means an array index can be scaled by the size of each array element.

The offset and base register can be used in three different ways to form the memory address. The addressing modes are described as follows:

Offset The offset is added to or subtracted from the base register to form the memory address.

Pre-indexed The offset is added to or subtracted from the base register to form the memory address. The base register is then updated with this new address, to permit automatic indexing through an array or memory block.

Post-indexed The value of the base register alone is used as the memory address. The offset is then added to or subtracted from the base register. The result is stored back in the base register, to permit automatic indexing through an array or memory block.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.6.3 非特権ロード / ストア

非特権モードでは、非特権のロード / ストアは対応する通常の操作と全く同じに動作します。特権モードでは、非特権ロード / ストアは非特権モードで実行されたものとして扱われます。詳細については、*データアクセスの特権レベルアクセス制御* P. A3- 38 を参照して下さい。

A4.6.4 排他ロード / ストア

排他ロード / ストアは共有メモリの同期に使用できます。詳細については、*同期化とセマフォ* P. A3- 12 を参照して下さい。

A4.6.5 アドレシングモード

ロード / ストアのアドレスは、ベースレジスタの値とオフセットの 2 つの部分から生成されます。

ベースレジスタには、任意の汎用レジスタが使用できます。

ロードの場合、PC をベースレジスタとすることができます。これによって、位置独立コードのための PC 相対アドレシングを使用できます。第 A8 章*「命令の詳細」*でタイトルに（リテラル）とマークされている命令は PC 相対ロードです。

オフセットは、次の 3 つのうちいずれかのフォーマットです。

イミディエート オフセットは符号なし数値で、ベースレジスタの値に加算または減算できます。イミディエートオフセット アドレシングは、データオブジェクトの先頭から一定の距離にあるデータエレメント、例えば構造体のフィールド、スタックオフセット、入力 / 出力レジスタなどにアクセスするのに便利です。

レジスタ オフセットは汎用レジスタの値です。このレジスタに PC は使用できません。この値はベースレジスタの値に加算または減算できます。レジスタオフセットは、配列やデータブロックにアクセスするのに便利です。

スケーリングされたレジスタ オフセットは PC 以外の汎用レジスタで、イミディエート値によってシフトされてから、ベースレジスタの値に加算または減算されます。つまり、配列のインデクスを、配列の各要素のサイズだけスケーリングできます。

オフセットとベースレジスタからメモリアドレスを計算するため、3 つの異なる方法を使用できます。このアドレシングモードは次に示すとおりです。

オフセット オフセットがベースレジスタに加算または減算され、メモリアドレスが計算されます。

プリインデクス オフセットがベースレジスタに加算または減算され、メモリアドレスが計算されます。その後で、ベースレジスタはその新しいアドレスに更新されるため、配列やメモリブロックを順番にアクセスする動作を自動的に実行できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

Not every variant is available for every instruction, and the range of permitted immediate values and the options for scaled registers vary from instruction to instruction. See Chapter A8 *Instruction Details* for full details for each instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ポストインデクス ベースレジスタの値が、そのままメモリアドレスとして使用されます。その後で、ベースレジスタにオフセットが加算または減算されます。結果はベースレジスタに書き戻されるため、配列やメモリブロックを順番にアクセスする動作を自動的に実行できます。

注

すべての命令についてすべてのバリアントが使用できるわけではありません。また、イミディエート値に使用できる値の範囲と、スケーリングされるレジスタのオプションは、命令によって異なります。各命令での詳細については、第 A8 章*「命令の詳細」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.7 Load/store multiple instructions

Load Multiple instructions load a subset, or possibly all, of the general-purpose registers from memory.

Store Multiple instructions store a subset, or possibly all, of the general-purpose registers to memory.

The memory locations are consecutive word-aligned words. The addresses used are obtained from a base register, and can be either above or below the value in the base register. The base register can optionally be updated by the total size of the data transferred.

Table A4-11 summarizes the load/store multiple instructions in the ARM and Thumb instruction sets.

Table A4-11 Load/store multiple instructions

Instruction	See
Load Multiple, Increment After or Full Descending	*LDM / LDMIA / LDMFD* on page A8-110
Load Multiple, Decrement After or Full Ascending [a]	*LDMDA / LDMFA* on page A8-112
Load Multiple, Decrement Before or Empty Ascending	*LDMDB / LDMEA* on page A8-114
Load Multiple, Increment Before or Empty Descending [a]	*LDMIB / LDMED* on page A8-116
Pop multiple registers off the stack [b]	*POP* on page A8-246
Push multiple registers onto the stack [c]	*PUSH* on page A8-248
Store Multiple, Increment After or Empty Ascending	*STM / STMIA / STMEA* on page A8-374
Store Multiple, Decrement After or Empty Descending [a]	*STMDA / STMED* on page A8-376
Store Multiple, Decrement Before or Full Descending	*STMDB / STMFD* on page A8-378
Store Multiple, Increment Before or Full Ascending [a]	*STMIB / STMFA* on page A8-380

a. Not available in the Thumb instruction set.
b. This instruction is equivalent to an `LDM` instruction with the SP as base register, and base register updating.
c. This instruction is equivalent to an `STMDB` instruction with the SP as base register, and base register updating.

System level variants of the `LDM` and `STM` instructions load and store User mode registers from a privileged mode. Another system level variant of the `LDM` instruction performs an exception return. For details, see Chapter B6 *System Instructions*.

A4.7.1 Loads to the PC

The `LDM`, `LDMDA`, `LDMDB`, `LDMIB`, and `POP` instructions can be used to load a value into the PC. The value loaded is treated as an interworking address, as described by the `LoadWritePC()` pseudocode function in *Pseudocode details of operations on ARM core registers* on page A2-12.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.7 複数ロード / ストア命令

複数ロード命令は、汎用レジスタのサブセットまたはすべてに、メモリからロードします。

複数ストア命令は、汎用レジスタのサブセットまたはすべてを、メモリへストアします。

メモリ位置は、連続したワードアラインドのワードです。使用されるアドレスはベースレジスタから取得され、ベースレジスタの値の上方または下方とすることができます。オプションとして、転送されたデータの合計サイズでベースレジスタを更新することもできます。

ARM および Thumb 命令セットの複数ロード / ストア命令の要約を、表 A4-11 に示します。

表 A4-11 複数ロード / ストア命令

命令	参照先
複数ロード（ポストインクリメントすなわちフル下降）	*LDM/LDMIA/LDMFD* P. A8-110
複数ロード（ポストデクリメントすなわちフル上昇）[a]	*LDMDA/LDMFA* P. A8-112
複数ロード（プリデクリメントすなわち空上昇）	*LDMDB/LDMEA* P. A8-114
複数ロード（プリインクリメントすなわち空下降）[a]	*LDMIB/LDMED* P. A8-116
複数のレジスタをスタックからポップ[b]	*POP* P. A8-246
複数のレジスタをスタックへプッシュ[c]	*PUSH* P. A8-248
複数ストア（ポストインクリメントすなわち空上昇）	*STM*、*STMIA*、*STMEA* P. A8-374
複数ストア（ポストデクリメントすなわち空下降）[a]	*STMDA*、*STMED* P. A8-376
複数ストア（プリデクリメントすなわちフル下降）	*STMDB*、*STMFD* P. A8-378
複数ストア（プリインクリメントすなわちフル上昇）[a]	*STMIB*、*STMFA* P. A8-380

a. Thumb 命令セットには存在しません。
b. この命令は、SP をベースレジスタとして使用しベースレジスタを更新する場合の `LDM` 命令と等価です。
c. この命令は、SP をベースレジスタとして使用しベースレジスタを更新する場合の `STMDB` 命令と等価です。

`LDM` および `STM` 命令のシステムレベル バリアントは、特権モードからユーザモード レジスタをロード / ストアします。`LDM` 命令の別のシステムレベル バリアントは、例外からの復帰を実行します。詳細については、第 B6 章「*システム命令*」を参照して下さい。

A4.7.1 PC へのロード

`LDM`、`LDMDA`、`LDMDB`、`LDMIB`、`POP` 命令は、値を PC へロードするために使用できます。ロードされた値は、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 の `LoadWritePC()` 擬似コード関数で説明されているように、インターワーキングアドレスとみなされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.8 Miscellaneous instructions

Table A4-12 summarizes the miscellaneous instructions in the ARM and Thumb instruction sets.

Table A4-12 Miscellaneous instructions

Instruction	See
Clear-Exclusive	*CLREX* on page A8-70
Debug hint	*DBG* on page A8-88
Data Memory Barrier	*DMB* on page A8-90
Data Synchronization Barrier	*DSB* on page A8-92
Instruction Synchronization Barrier	*ISB* on page A8-102
If Then (makes following instructions conditional)	*IT* on page A8-104
No Operation	*NOP* on page A8-222
Preload Data	*PLD, PLDW (immediate)* on page A8-236 *PLD (literal)* on page A8-238 *PLD, PLDW (register)* on page A8-240
Preload Instruction	*PLI (immediate, literal)* on page A8-242 *PLI (register)* on page A8-244
Set Endianness	*SETEND* on page A8-314
Send Event	*SEV* on page A8-316
Supervisor Call	*SVC (previously SWI)* on page A8-430
Swap, Swap Byte. Use deprecated. [a]	*SWP, SWPB* on page A8-432
Wait For Event	*WFE* on page A8-808
Wait For Interrupt	*WFI* on page A8-810
Yield	*YIELD* on page A8-812

a. Use Load&Store-Exclusive instructions instead, see *Load/store instructions* on page A4-19.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.8 その他の命令

ARM 命令セットおよび Thumb 命令セットのその他の命令の要約を、表 A4-12 に示します。

表 A4-12 その他の命令

命令	参照先
排他クリア	*CLREX* P. A8-70
デバッグヒント	*DBG* P. A8-88
データ メモリバリア	*DMB* P. A8-90
データ同期バリア	*DSB* P. A8-92
命令同期バリア	*ISB* P. A8-102
If Then（以後の命令を条件付きにする）	*IT* P. A8-104
操作なし	*NOP* P. A8-222
データのプリロード	*PLD*、*PLDW（イミディエート）* P. A8-236 *PLD（リテラル）* P. A8-238 *PLD*、*PLDW（レジスタ）* P. A8-240
命令のプリロード	*PLI（イミディエート、リテラル）* P. A8-242 *PLI（レジスタ）* P. A8-244
エンディアン設定	*SETEND* P. A8-314
イベント送信	*SEV* P. A8-316
スーパバイザコール	*SVC（以前の SWI）* P. A8-430
スワップ、バイトスワップ。使用は推奨されません。[a]	*SWP*、*SWPB* P. A8-432
イベント待ち	*WFE* P. A8-808
割り込み待ち	*WFI* P. A8-810
イールド	*YIELD* P. A8-812

a. 代わりに排他ロード / 排他ストア命令を使用します（*ロード / ストア命令* P. A4-19 参照）。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.9 Exception-generating and exception-handling instructions

The following instructions are intended specifically to cause a processor exception to occur:

- The Supervisor Call (SVC, previously SWI) instruction is used to cause an SVC exception to occur. This is the main mechanism for User mode code to make calls to privileged operating system code. For more information, see *Supervisor Call (SVC) exception* on page B1-52.
- The Breakpoint instruction BKPT provides for software breakpoints. For more information, see *About debug events* on page C3-2.
- In privileged system level code, the Secure Monitor Call (SMC, previously SMI) instruction. For more information, see *Secure Monitor Call (SMC) exception* on page B1-53.

System level variants of the SUBS and LDM instructions can be used to return from exceptions. From ARMv6, the SRS instruction can be used near the start of an exception handler to store return information, and the RFE instruction can be used to return from an exception using the stored return information. For details of these instructions, see Chapter B6 *System Instructions*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.9 例外生成命令と例外処理命令

次の命令は、プロセッサ例外を生成することを特に目的としている命令です。

- スーパバイザコール（SVC、以前の SWI）命令は、SVC 例外を起こすために使用されます。これは、ユーザモードのコードが特権オペレーティングシステム コードを呼び出すための主要な機構です。詳細については、*スーパバイザコール (SVC) 例外* P. B1- 52 を参照して下さい。
- ブレークポイント命令の BKPT は、ソフトウェアブレークポイントを提供します。詳細については、*デバッグイベントについて*P. C3- 2 を参照して下さい。
- 特権システムレベルコードのセキュア モニタコール（SMC、以前の SMI）命令。詳細については、*セキュア モニタコール (SMC) 例外* P. B1- 53 を参照して下さい。

SUBS および LDM 命令のシステムレベル バリアントは、例外からの復帰に使用できます。ARMv6 およびそれ以降では、SRS 命令を例外ハンドラの先頭付近で使用し、復帰情報をストアすることができます。そして、ストアした復帰情報を使って RFE 命令で例外から復帰することができます。これらの命令の詳細については、第 B6 章*「システム命令」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.10 Coprocessor instructions

There are three types of instruction for communicating with coprocessors. These permit the processor to:

- Initiate a coprocessor data-processing operation. For details see *CDP, CDP2* on page A8-68.
- Transfer general-purpose registers to and from coprocessor registers. For details, see:
  - *MCR, MCR2* on page A8-186
  - *MCRR, MCRR2* on page A8-188
  - *MRC, MRC2* on page A8-202
  - *MRRC, MRRC2* on page A8-204.
- Load or store the values of coprocessor registers. For details, see:
  - *LDC, LDC2 (immediate)* on page A8-106
  - *LDC, LDC2 (literal)* on page A8-108
  - *STC, STC2* on page A8-372.

The instruction set distinguishes up to 16 coprocessors with a 4-bit field in each coprocessor instruction, so each coprocessor is assigned a particular number.

Note

One coprocessor can use more than one of the 16 numbers if a large coprocessor instruction set is required.

Coprocessors 10 and 11 are used, together, for VFP and some Advanced SIMD functionality. There are different instructions for accessing these coprocessors, of similar types to the instructions for the other coprocessors, that is, to:

- Initiate a coprocessor data-processing operation. For details see *VFP data-processing instructions* on page A4-38.
- Transfer general-purpose registers to and from coprocessor registers. For details, see *Advanced SIMD and VFP register transfer instructions* on page A4-29.
- Load or store the values of coprocessor registers. For details, see *Advanced SIMD and VFP load/store instructions* on page A4-26.

Coprocessors execute the same instruction stream as the processor, ignoring non-coprocessor instructions and coprocessor instructions for other coprocessors. Coprocessor instructions that cannot be executed by any coprocessor hardware cause an Undefined Instruction exception.

For more information about specific coprocessors see *Coprocessor support* on page A2-68.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.10 コプロセッサ命令

コプロセッサと通信するための命令には 3 つのタイプがあります。これらの命令により、プロセッサは次の動作を実行できます。

- コプロセッサのデータ処理動作を開始する。詳細については、*CDP*、*CDP2* P. A8-68 を参照して下さい。
- 汎用レジスタとコプロセッサレジスタとの間でデータを転送する。詳細については、次の項目を参照して下さい。
  - *MCR*、*MCR2* P. A8-186
  - *MCRR*、*MCRR2* P. A8-188
  - *MRC*、*MRC2* P. A8-202
  - *MRRC*、*MRRC2* P. A8-204
- コプロセッサレジスタの値をロードまたはストアする。詳細については、次の項目を参照して下さい。
  - *LDC*、*LDC2（イミディエート）* P. A8-106
  - *LDC*、*LDC2（リテラル）* P. A8-108
  - *STC*、*STC2* P. A8-372

命令セットでは、各コプロセッサ命令に含まれている 4 ビットのフィールドで最大 16 のコプロセッサを区別でき、各コプロセッサに固有の番号が割り当てられます。

注

大規模なコプロセッサ命令セットが必要な場合、1 つのコプロセッサで 16 個の番号のうち 2 つ以上を使用できます。

コプロセッサ 10 および 11 は組として、VFP および一部のアドバンスト SIMD 機能に用されます。これらのコプロセッサにアクセスするため、他のコプロセッサの場合と同様のタイプの命令が存在します。これらの命令により、次の動作を実行できます。

- コプロセッサのデータ処理動作を開始する。詳細については、*VFP のデータ処理命令* P. A4-38 を参照して下さい。
- 汎用レジスタとコプロセッサレジスタとの間でデータを転送する。詳細については、*アドバンスト SIMD および VFP のレジスタ転送命令* P. A4-29 を参照して下さい。
- コプロセッサレジスタの値をロードまたはストアする。詳細については、*アドバンスト SIMD および VFP のロード／ストア命令* P. A4-26 を参照して下さい。

コプロセッサは、プロセッサと同じ命令ストリームを実行し、コプロセッサ命令でない命令や、他のコプロセッサ用の命令は無視します。どのコプロセッサハードウェアでも実行できないコプロセッサ命令は、未定義命令例外を引き起こします。

各コプロセッサの詳細については、*コプロセッサのサポート* P. A2-68 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.11 Advanced SIMD and VFP load/store instructions

Table A4-13 summarizes the extension register load/store instructions in the Advanced SIMD and VFP instruction sets.

Advanced SIMD also provides instructions for loading and storing multiple elements, or structures of elements, see *Element and structure load/store instructions* on page A4-27.

Table A4-13 Extension register load/store instructions

Instruction	See	Operation
Vector Load Multiple	*VLDM* on page A8-626	Load 1-16 consecutive 64-bit registers (Adv. SIMD and VFP) Load 1-16 consecutive 32-bit registers (VFP only)
Vector Load Register	*VLDR* on page A8-628	Load one 64-bit register (Adv. SIMD and VFP) Load one 32-bit register (VFP only)
Vector Store Multiple	*VSTM* on page A8-784	Store 1-16 consecutive 64-bit registers (Adv. SIMD and VFP) Store 1-16 consecutive 32-bit registers (VFP only)
Vector Store Register	*VSTR* on page A8-786	Store one 64-bit register (Adv. SIMD and VFP) Store one 32-bit register (VFP only)

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.11 アドバンスト SIMD および VFP のロード / ストア命令

アドバンスト SIMD および VFP 命令セットに含まれる拡張機能レジスタのロード / ストア命令の要約を、表 A4-13 に示します。

アドバンスト SIMD も、複数のエレメントまたはエレメントの構造をロード / ストアする命令を備えています。*エレメントおよび構造のロード / ストア命令* P. A4-27 を参照して下さい。

表 A4-13 拡張機能レジスタロード / ストア命令

命令	参照先	動作
ベクタ複数ロード	*VLDM* P. A8-626	1 ～ 16 個の連続する 64 ビットレジスタをロード（アドバンスト SIMD および VFP） 1 ～ 16 個の連続する 32 ビットレジスタをロード（VFP のみ）
ベクタ レジスタロード	*VLDR* P. A8-628	1 つの 64 ビットレジスタをロード（アドバンスト SIMD および VFP） 1 つの 32 ビットレジスタをロード（VFP のみ）
ベクタ複数ストア	*VSTM* P. A8-784	1 ～ 16 個の連続する 64 ビットレジスタをストア（アドバンスト SIMD および VFP） 1 ～ 16 個の連続する 32 ビットレジスタをストア（VFP のみ）
ベクタ レジスタストア	*VSTR* P. A8-786	1 つの 64 ビットレジスタをストア（アドバンスト SIMD および VFP） 1 つの 32 ビットレジスタをストア（VFP のみ）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.11.1 Element and structure load/store instructions

Table A4-14 shows the element and structure load/store instructions available in the Advanced SIMD instruction set. Loading and storing structures of more than one element automatically de-interleaves or interleaves the elements, see Figure A4-1 on page A4-28 for an example of de-interleaving. Interleaving is the inverse process.

Table A4-14 Element and structure load/store instructions

Instruction	See
Load single element	
Multiple elements	*VLD1 (multiple single elements)* on page A8-602
To one lane	*VLD1 (single element to one lane)* on page A8-604
To all lanes	*VLD1 (single element to all lanes)* on page A8-606
Load 2-element structure	
Multiple structures	*VLD2 (multiple 2-element structures)* on page A8-608
To one lane	*VLD2 (single 2-element structure to one lane)* on page A8-610
To all lanes	*VLD2 (single 2-element structure to all lanes)* on page A8-612
Load 3-element structure	
Multiple structures	*VLD3 (multiple 3-element structures)* on page A8-614
To one lane	*VLD3 (single 3-element structure to one lane)* on page A8-616
To all lanes	*VLD3 (single 3-element structure to all lanes)* on page A8-618
Load 4-element structure	
Multiple structures	*VLD4 (multiple 4-element structures)* on page A8-620
To one lane	*VLD4 (single 4-element structure to one lane)* on page A8-622
To all lanes	*VLD4 (single 4-element structure to all lanes)* on page A8-624
Store single element	
Multiple elements	*VST1 (multiple single elements)* on page A8-768
From one lane	*VST1 (single element from one lane)* on page A8-770

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.11.1 エレメントおよび構造のロード / ストア命令

アドバンスト SIMD 命令セットに含まれるエレメントおよび構造のロード / ストア命令の要約を、表 A4-14 に示します。複数のエレメントで構成される構造のロード / ストアでは、エレメントのインターリーブ解除またはインターリーブが自動的に行われます。インターリーブ解除の例については、P. A4-28 の図 A4-1 を参照して下さい。インターリーブは反転プロセスです。

表 A4-14 エレメントおよび構造のロード / ストア命令

命令	参照先
単一エレメントのロード	
複数のエレメント	*VLD1（複数の単一エレメントのロード）* P. A8-602
1 レーンへ	*VLD1（1 つのレーンへの単一エレメントのロード）* P. A8-604
すべてのレーンへ	*VLD1（全レーンへの単一エレメントのロード）* P. A8-606
2 エレメント構造のロード	
複数の構造	*VLD2（複数の 2 エレメント構造のロード）* P. A8-608
1 レーンへ	*VLD2（1 つのレーンへの単一の 2 エレメント構造のロード）* P. A8-610
すべてのレーンへ	*VLD2（全レーンへの単一の 2 エレメント構造のロード）* P. A8-612
3 エレメント構造のロード	
複数の構造	*VLD3（複数の 3 エレメント構造のロード）* P. A8-614
1 レーンへ	*VLD3（1 つのレーンへの単一の 3 エレメント構造のロード）* P. A8-616
すべてのレーンへ	*VLD3（全レーンへの単一の 3 エレメント構造のロード）* P. A8-618
4 エレメント構造のロード	
複数の構造	*VLD4（複数の 4 エレメント構造のロード）* P. A8-620
1 レーンへ	*VLD4（1 つのレーンへの単一の 4 エレメント構造のロード）* P. A8-622
すべてのレーンへ	*VLD4（全レーンへの単一の 4 エレメント構造のロード）* P. A8-624
単一エレメントのストア	
複数のエレメント	*VST1（複数の単一エレメントのストア）* P. A8-768
1 レーンから	*VST1（1 つのレーンからの単一エレメントのストア）* P. A8-770

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A4-14 Element and structure load/store instructions (continued)

Instruction	See
Store 2-element structure	
Multiple structures	*VST2 (multiple 2-element structures)* on page A8-772
From one lane	*VST2 (single 2-element structure from one lane)* on page A8-774
Store 3-element structure	
Multiple structures	*VST3 (multiple 3-element structures)* on page A8-776
From one lane	*VST3 (single 3-element structure from one lane)* on page A8-778
Store 4-element structure	
Multiple structures	*VST4 (multiple 4-element structures)* on page A8-780
From one lane	*VST4 (single 4-element structure from one lane)* on page A8-782

Figure A4-1 De-interleaving an array of 3-element structures

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A4-14 エレメントおよび構造のロード / ストア命令（続き）

命令	参照先
2 エレメント構造のストア	
複数の構造	*VST2（複数の 2 エレメント構造のストア）* P. A8-772
1 レーンから	*VST2（1 つのレーンからの単一の 2 エレメント構造のストア）* P. A8-774
3 エレメント構造のストア	
複数の構造	*VST3（複数の 3 エレメント構造のストア）* P. A8-776
1 レーンから	*VST3（1 つのレーンからの単一の 3 エレメント構造のストア）* P. A8-778
4 エレメント構造のストア	
複数の構造	*VST4（複数の 4 エレメント構造のストア）* P. A8-780
1 レーンから	*VST4（1 つのレーンからの単一の 4 エレメント構造のストア）* P. A8-782

図 A4-1 3 エレメント構造の配列のインターリーブ解除

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.12 Advanced SIMD and VFP register transfer instructions

Table A4-15 summarizes the extension register transfer instructions in the Advanced SIMD and VFP instruction sets. These instructions transfer data from ARM core registers to extension registers, or from extension registers to ARM core registers.

Advanced SIMD vectors, and single-precision and double-precision VFP registers, are all views of the same extension register set. For details see *Advanced SIMD and VFP extension registers* on page A2-21.

Table A4-15 Extension register transfer instructions

Instruction	See
Copy element from ARM core register to every element of Advanced SIMD vector	*VDUP (ARM core register)* on page A8-594
Copy byte, halfword, or word from ARM core register to extension register	*VMOV (ARM core register to scalar)* on page A8-644
Copy byte, halfword, or word from extension register to ARM core register	*VMOV (scalar to ARM core register)* on page A8-646
Copy from single-precision VFP register to ARM core register, or from ARM core register to single-precision VFP register	*VMOV (between ARM core register and single-precision register)* on page A8-648
Copy two words from ARM core registers to consecutive single-precision VFP registers, or from consecutive single-precision VFP registers to ARM core registers	*VMOV (between two ARM core registers and two single-precision registers)* on page A8-650
Copy two words from ARM core registers to doubleword extension register, or from doubleword extension register to ARM core registers	*VMOV (between two ARM core registers and a doubleword extension register)* on page A8-652
Copy from Advanced SIMD and VFP extension System Register to ARM core register	*VMRS* on page A8-658 *VMRS* on page B6-27 (system level view)
Copy from ARM core register to Advanced SIMD and VFP extension System Register	*VMSR* on page A8-660 *VMSR* on page B6-29 (system level view)

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.12 アドバンスト SIMD および VFP のレジスタ転送命令

アドバンスト SIMD および VFP 命令セットに含まれる拡張機能レジスタ転送命令の要約を、表 A4-15 に示します。これらの命令は、ARM コアレジスタから拡張機能レジスタへ、または拡張機能レジスタから ARM コアレジスタへデータを転送します。

アドバンスト SIMD ベクタ、および単精度、倍精度 VFP レジスタは、すべて同じ拡張機能レジスタセットのビューです。詳細については、*アドバンスト SIMD と VFP の拡張機能レジスタ* P. A2-21 を参照して下さい。

表 A4-15 拡張機能レジスタ転送命令

命令	参照先
ARM コアレジスタからアドバンスト SIMD ベクタのすべてのエレメントへエレメントをコピー	*VDUP（ARM コアレジスタ）* P. A8-594
ARM コアレジスタから拡張機能レジスタへバイト、ハーフワード、またはワードをコピー	*VMOV（ARM コアレジスタからスカラへのコピー）* P. A8-644
拡張機能レジスタから ARM コアレジスタへバイト、ハーフワード、またはワードをコピー	*VMOV（スカラから ARM コアレジスタへのコピー）* P. A8-646
単精度 VFP レジスタから ARM コアレジスタへ、または ARM コアレジスタから単精度 VFP レジスタへコピー	*VMOV（ARM コアレジスタと単精度レジスタとの間の転送）* P. A8-648
ARM コアレジスタから連続する単精度 VFP レジスタへ、または連続する単精度 VFP レジスタから ARM コアレジスタへ、2 ワードをコピー	*VMOV（2 つの ARM コアレジスタと 2 つの単精度レジスタとの間の転送）* P. A8-650
ARM コアレジスタからダブルワード拡張機能レジスタへ、またはダブルワード拡張機能レジスタから ARM コアレジスタへ、2 ワードをコピー	*VMOV（2 つの ARM コアレジスタとダブルワード拡張機能レジスタとの間のコピー）* P. A8-652
アドバンスト SIMD および VFP 拡張機能システムレジスタから ARM コアレジスタへコピー	*VMRS* P. A8-658 *VMRS* P. B6-27（システムレベルのビュー）
ARM コアレジスタからアドバンスト SIMD および VFP 拡張機能システムレジスタへコピー	*VMSR* P. A8-660 *VMSR* P. B6-29（システムレベルのビュー）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.13 Advanced SIMD data-processing operations

Advanced SIMD data-processing operations process registers containing vectors of elements of the same type packed together, enabling the same operation to be performed on multiple items in parallel.

Instructions operate on vectors held in 64-bit or 128-bit registers. Figure A4-2 shows an operation on two 64-bit operand vectors, generating a 64-bit vector result.

Note

Figure A4-2 and other similar figures show 64-bit vectors that consist of four 16-bit elements, and 128-bit vectors that consist of four 32-bit elements. Other element sizes produce similar figures, but with one, two, eight, or sixteen operations performed in parallel instead of four.

Figure A4-2 Advanced SIMD instruction operating on 64-bit registers

Many Advanced SIMD instructions have variants that produce vectors of elements double the size of the inputs. In this case, the number of elements in the result vector is the same as the number of elements in the operand vectors, but each element, and the whole vector, is double the size.

Figure A4-3 shows an example of an Advanced SIMD instruction operating on 64-bit registers, and generating a 128-bit result.

Figure A4-3 Advanced SIMD instruction producing wider result

There are also Advanced SIMD instructions that have variants that produce vectors containing elements half the size of the inputs. Figure A4-4 on page A4-31 shows an example of an Advanced SIMD instruction operating on one 128-bit register, and generating a 64-bit result.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.13 アドバンスト SIMD のデータ処理操作

アドバンスト SIMD のデータ処理操作では、同じタイプのエレメントのベクタをパックしたレジスタを処理することにより、同じ操作を複数のアイテムに並列的に実行できます。

命令は、64 ビットまたは 128 ビットレジスタに保持されたベクタに作用します。2 つの 64 ビットオペランド ベクタから 1 つの 64 ビットベクタの結果を生成する動作を、図 A4-2 に示します。

注

図 A4-2 および他の同様の図は、4 つの 16 ビットエレメントで構成される 64 ビットベクタ、および 4 つの 32 ビットエレメントで構成される 128 ビットベクタを示しています。他のエレメントサイズでも同様の図になりますが、4 つではなく、1、2、8、16 個の演算が並列して実行されます。

図 A4-2 64 ビットレジスタに作用するアドバンスト SIMD 命令

多くのアドバンスト SIMD 命令には、入力の 2 倍のサイズを持つエレメントのベクタを生成するバリアントがあります。この場合、結果となるベクタ内のエレメント数はオペランドベクタ内のエレメントの数と同じですが、各エレメントとベクタ全体はサイズが 2 倍になっています。

64 ビットレジスタに作用し、128 ビットの結果を生成するアドバンスト SIMD 命令の例を、図 A4-3 に示します。

図 A4-3 幅の広い結果を生成するアドバンスト SIMD 命令

一部のアドバンスト SIMD 命令には、入力の半分のサイズを持つエレメントのベクタを生成するバリアントも存在します。1 つの 128 ビットレジスタに作用し、64 ビットの結果を生成するアドバンスト SIMD 命令の例を、P. A4-31 の図 A4-4 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure A4-4 Advanced SIMD instruction producing narrower result

Some Advanced SIMD instructions do not conform to these standard patterns. Their operation patterns are described in the individual instruction descriptions.

Advanced SIMD instructions that perform floating-point arithmetic use the ARM standard floating-point arithmetic defined in *Floating-point data types and arithmetic* on page A2-32.

A4.13.1 Advanced SIMD parallel addition and subtraction

Table A4-16 shows the Advanced SIMD parallel add and subtract instructions.

Table A4-16 Advanced SIMD parallel add and subtract instructions

Instruction	See
Vector Add	*VADD (integer)* on page A8-536 *VADD (floating-point)* on page A8-538
Vector Add and Narrow, returning High Half	*VADDHN* on page A8-540
Vector Add Long, Vector Add Wide	*VADDL, VADDW* on page A8-542
Vector Halving Add, Vector Halving Subtract	*VHADD, VHSUB* on page A8-600
Vector Pairwise Add and Accumulate Long	*VPADAL* on page A8-682
Vector Pairwise Add	*VPADD (integer)* on page A8-684 *VPADD (floating-point)* on page A8-686
Vector Pairwise Add Long	*VPADDL* on page A8-688
Vector Rounding Add and Narrow, returning High Half	*VRADDHN* on page A8-726
Vector Rounding Halving Add	*VRHADD* on page A8-734
Vector Rounding Subtract and Narrow, returning High Half	*VRSUBHN* on page A8-748
Vector Saturating Add	*VQADD* on page A8-700
Vector Saturating Subtract	*VQSUB* on page A8-724

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

図 A4-4 幅の狭い結果を生成するアドバンスト SIMD 命令

この標準的なパターンに従わないアドバンスト SIMD 命令もあります。これらの命令の動作パターンは、各命令の説明に記載されています。

浮動小数点算術演算を実行するアドバンスト SIMD 命令は、*浮動小数点データ型と算術演算* P. A2-32 に定義されている ARM 標準浮動小数点算術演算を使用します。

A4.13.1 アドバンスト SIMD の並列加算および減算

アドバンスト SIMD の並列加算および減算命令を、表 A4-16 に示します。

表 A4-16 アドバンスト SIMD の並列加算および減算命令

命令	参照先
ベクタ加算	*VADD（整数）* P. A8-536 *VADD（浮動小数点）* P. A8-538
上位半分を返すベクタ加算および縮小	*VADDHN* P. A8-540
ベクタロング加算、ベクタワイド加算	*VADDL*、*VADDW* P. A8-542
ベクタ加算半減、ベクタ減算半減	*VHADD*、*VHSUB* P. A8-600
ベクタペア単位ロング加算および累積	*VPADAL* P. A8-682
ベクタペア単位加算	*VPADD（整数）* P. A8-684 *VPADD（浮動小数点数）* P. A8-686
ベクタペア単位ロング加算	*VPADDL* P. A8-688
上位半分を返すベクタ丸め加算および縮小	*VRADDHN* P. A8-726
ベクタ丸め加算半減	*VRHADD* P. A8-734
上位半分を返すベクタ丸め減算および縮小	*VRSUBHN* P. A8-748
ベクタ飽和加算	*VQADD* P. A8-700
ベクタ飽和減算	*VQSUB* P. A8-724

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A4-16 Advanced SIMD parallel add and subtract instructions (continued)

Instruction	See
Vector Subtract	*VSUB (integer)* on page A8-788 *VSUB (floating-point)* on page A8-790
Vector Subtract and Narrow, returning High Half	*VSUBHN* on page A8-792
Vector Subtract Long, Vector Subtract Wide	*VSUBL, VSUBW* on page A8-794

A4.13.2 Bitwise Advanced SIMD data-processing instructions

Table A4-17 shows bitwise Advanced SIMD data-processing instructions. These operate on the doubleword (64-bit) or quadword (128-bit) extension registers, and there is no division into vector elements.

Table A4-17 Bitwise Advanced SIMD data-processing instructions

Instruction	See
Vector Bitwise AND	*VAND (register)* on page A8-544
Vector Bitwise Bit Clear (AND complement)	*VBIC (immediate)* on page A8-546 *VBIC (register)* on page A8-548
Vector Bitwise Exclusive OR	*VEOR* on page A8-596
Vector Bitwise Insert if False	*VBIF, VBIT, VBSL* on page A8-550
Vector Bitwise Insert if True	
Vector Bitwise Move	*VMOV (immediate)* on page A8-640 *VMOV (register)* on page A8-642
Vector Bitwise NOT	*VMVN (immediate)* on page A8-668 *VMVN (register)* on page A8-670
Vector Bitwise OR	*VORR (immediate)* on page A8-678 *VORR (register)* on page A8-680
Vector Bitwise OR NOT	*VORN (register)* on page A8-676
Vector Bitwise Select	*VBIF, VBIT, VBSL* on page A8-550

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A4-16 アドバンスト SIMD の並列加算および減算命令（続き）

命令	参照先
ベクタ減算	*VSUB（整数）* P. A8-788 *VSUB（浮動小数点数）* P. A8-790
上位半分を返すベクタ減算および縮小	*VSUBHN* P. A8-792
ベクタロング減算、ベクタワイド減算	*VSUBL*、*VSUBW* P. A8-794

A4.13.2 アドバンスト SIMD のビット単位データ処理命令

アドバンスト SIMD のビット単位データ処理命令を、表 A4-17 に示します。これらは、ダブルワード（64 ビット）またはクワッドワード（128 ビット）の拡張機能レジスタに作用し、ベクタエレメントへの分割はありません。

表 A4-17 アドバンスト SIMD のビット単位データ処理命令

命令	参照先
ベクタビット単位論理積	*VAND（レジスタ）* P. A8-544
ベクタビット単位ビットクリア（補数論理積）	*VBIC（イミディエート）* P. A8-546 *VBIC（レジスタ）* P. A8-548
ベクタビット単位排他的論理和	*VEOR* P. A8-596
FALSE 時ベクタビット単位挿入	*VBIF*、*VBIT*、*VBSL* P. A8-550
TRUE 時ベクタビット単位挿入	
ベクタビット単位移動	*VMOV（イミディエート）* P. A8-640 *VMOV（レジスタ）* P. A8-642
ベクタビット単位否定	*VMVN（イミディエート）* P. A8-668 *VMVN（レジスタ）* P. A8-670
ベクタビット単位論理和	*VORR（イミディエート）* P. A8-678 *VORR（レジスタ）* P. A8-680
ベクタビット単位否定論理和	*VORN（レジスタ）* P. A8-676
ベクタビット単位選択	*VBIF*、*VBIT*、*VBSL* P. A8-550

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.13.3 Advanced SIMD comparison instructions

Table A4-18 shows Advanced SIMD comparison instructions.

Table A4-18 Advanced SIMD comparison instructions

Instruction	See
Vector Absolute Compare	*VACGE, VACGT, VACLE,VACLT* on page A8-534
Vector Compare Equal	*VCEQ (register)* on page A8-552
Vector Compare Equal to Zero	*VCEQ (immediate #0)* on page A8-554
Vector Compare Greater Than or Equal	*VCGE (register)* on page A8-556
Vector Compare Greater Than or Equal to Zero	*VCGE (immediate #0)* on page A8-558
Vector Compare Greater Than	*VCGT (register)* on page A8-560
Vector Compare Greater Than Zero	*VCGT (immediate #0)* on page A8-562
Vector Compare Less Than or Equal to Zero	*VCLE (immediate #0)* on page A8-564
Vector Compare Less Than Zero	*VCLT (immediate #0)* on page A8-568
Vector Test Bits	*VTST* on page A8-802

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.13.3 アドバンスト SIMD の比較命令

アドバンスト SIMD の比較命令を、表 A4-18 に示します。

表 A4-18 アドバンスト SIMD の比較命令

命令	参照先
ベクタ絶対値比較	*VACGE*、*VACGT*、*VACLE*、*VACLT* P. A8-534
ベクタ比較等価	*VCEQ（レジスタ）* P. A8-552
ベクタ比較ゼロ等価	*VCEQ（イミディエート #0）* P. A8-554
ベクタ比較以上	*VCGE（レジスタ）* P. A8-556
ベクタ比較ゼロ以上	*VCGE（イミディエート #0）* P. A8-558
ベクタ比較より大きい	*VCGT（レジスタ）* P. A8-560
ベクタ比較ゼロより大きい	*VCGT（イミディエート #0）* P. A8-562
ベクタ比較ゼロ以下	*VCLE（イミディエート #0）* P. A8-564
ベクタ比較ゼロより小さい	*VCLT（イミディエート #0）* P. A8-568
ベクタ テストビット	*VTST* P. A8-802

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.13.4 Advanced SIMD shift instructions

Table A4-19 lists the shift instructions in the Advanced SIMD instruction set.

Table A4-19 Advanced SIMD shift instructions

Instruction	See
Vector Saturating Rounding Shift Left	*VQRSHL* on page A8-714
Vector Saturating Rounding Shift Right and Narrow	*VQRSHRN, VQRSHRUN* on page A8-716
Vector Saturating Shift Left	*VQSHL (register)* on page A8-718 *VQSHL, VQSHLU (immediate)* on page A8-720
Vector Saturating Shift Right and Narrow	*VQSHRN, VQSHRUN* on page A8-722
Vector Rounding Shift Left	*VRSHL* on page A8-736
Vector Rounding Shift Right	*VRSHR* on page A8-738
Vector Rounding Shift Right and Accumulate	*VRSRA* on page A8-746
Vector Rounding Shift Right and Narrow	*VRSHRN* on page A8-740
Vector Shift Left	*VSHL (immediate)* on page A8-750 *VSHL (register)* on page A8-752
Vector Shift Left Long	*VSHLL* on page A8-754
Vector Shift Right	*VSHR* on page A8-756
Vector Shift Right and Narrow	*VSHRN* on page A8-758
Vector Shift Left and Insert	*VSLI* on page A8-760
Vector Shift Right and Accumulate	*VSRA* on page A8-764
Vector Shift Right and Insert	*VSRI* on page A8-766

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.13.4 アドバンスト SIMD のシフト命令

アドバンスト SIMD 命令セットのシフト命令の一覧を、表 A4-19 に示します。

表 A4-19 アドバンスト SIMD のシフト命令

命令	参照先
ベクタ飽和丸め付き左シフト	*VQRSHL* P. A8-714
ベクタ飽和丸め付き右シフトおよび縮小	*VQRSHRN*、*VQRSHRUN* P. A8-716
ベクタ飽和左シフト	*VQSHL（レジスタ）* P. A8-718 *VQSHL*、*VQSHLU（イミディエート）* P. A8-720
ベクタ飽和右シフトおよび縮小	*VQSHRN*、*VQSHRUN* P. A8-722
ベクタ丸め付き左シフト	*VRSHL* P. A8-736
ベクタ丸め付き右シフト	*VRSHR* P. A8-738
ベクタ丸め付き右シフトおよび累積	*VRSRA* P. A8-746
ベクタ丸め付き右シフトおよび縮小	*VRSHRN* P. A8-740
ベクタ左シフト	*VSHL（イミディエート）* P. A8-750 *VSHL（レジスタ）* P. A8-752
ベクタ左シフトロング	*VSHLL* P. A8-754
ベクタ右シフト	*VSHR* P. A8-756
ベクタ右シフトおよび縮小	*VSHRN* P. A8-758
ベクタ左シフトおよび挿入	*VSLI* P. A8-760
ベクタ右シフトおよび累積	*VSRA* P. A8-764
ベクタ右シフトおよび挿入	*VSRI* P. A8-766

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.13.5 Advanced SIMD multiply instructions

Table A4-20 summarizes the Advanced SIMD multiply instructions.

Table A4-20 Advanced SIMD multiply instructions

Instruction	See
Vector Multiply Accumulate	*VMLA, VMLAL, VMLS, VMLSL (integer)* on page A8-634 *VMLA, VMLS (floating-point)* on page A8-636 *VMLA, VMLAL, VMLS, VMLSL (by scalar)* on page A8-638
Vector Multiply Accumulate Long	
Vector Multiply Subtract	
Vector Multiply Subtract Long	
Vector Multiply	*VMUL, VMULL (integer and polynomial)* on page A8-662 *VMUL (floating-point)* on page A8-664 *VMUL, VMULL (by scalar)* on page A8-666
Vector Multiply Long	
Vector Saturating Doubling Multiply Accumulate Long	*VQDMLAL, VQDMLSL* on page A8-702
Vector Saturating Doubling Multiply Subtract Long	
Vector Saturating Doubling Multiply Returning High Half	*VQDMULH* on page A8-704
Vector Saturating Rounding Doubling Multiply Returning High Half	*VQRDMULH* on page A8-712
Vector Saturating Doubling Multiply Long	*VQDMULL* on page A8-706

Advanced SIMD multiply instructions can operate on vectors of:

- 8-bit, 16-bit, or 32-bit unsigned integers
- 8-bit, 16-bit, or 32-bit signed integers
- 8-bit or 16-bit polynomials over {0,1} (`VMUL` and `VMULL` only)
- single-precision (32-bit) floating-point numbers.

They can also act on one vector and one scalar.

Long instructions have doubleword (64-bit) operands, and produce quadword (128-bit) results. Other Advanced SIMD multiply instructions can have either doubleword or quadword operands, and produce results of the same size.

VFP multiply instructions can operate on:

- single-precision (32-bit) floating-point numbers
- double-precision (64-bit) floating-point numbers.

Some VFP implementations do not support double-precision numbers.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.13.5 アドバンスト SIMD の乗算命令

アドバンスト SIMD の乗算命令の要約を、表 A4-20 に示します。

表 A4-20 アドバンスト SIMD の乗算命令

命令	参照先
ベクタ積和	*VMLA*、*VMLAL*、*VMLS*、*VMLSL（整数）* P. A8-634 *VMLA*、*VMLS（浮動小数点数）* P. A8-636 *VMLA*、*VMLAL*、*VMLS*、*VMLSL（スカラ）* P. A8-638
ベクタロング積和	
ベクタ積減算	
ベクタロング積減算	
ベクタ乗算	*VMUL*、*VMULL（整数および多項式）* P. A8-662 *VMUL*、*VMULL（浮動小数点数）* P. A8-664 *VMUL*、*VMULL（スカラ）* P. A8-666
ベクタロング乗算	
ベクタ飽和ロング積和 2 倍	*VQDMLAL*、*VQDMLSL* P. A8-702
ベクタ飽和ロング積減算 2 倍	
上位半分を返すベクタ飽和乗算 2 倍	*VQDMULH* P. A8-704
上位半分を返すベクタ飽和丸め付き乗算 2 倍	*VQRDMULH* P. A8-712
ベクタ飽和ロング乗算 2 倍	*VQDMULL* P. A8-706

アドバンスト SIMD の乗算命令は、次のベクタに作用します。

- 8 ビット、16 ビット、または 32 ビットの符号なし整数
- 8 ビット、16 ビット、または 32 ビットの符号付き整数
- {0,1} における 8 ビットまたは 16 ビットの多項式（`VMUL` および `VMULL` のみ）
- 単精度（32 ビット）の浮動小数点数

1 つのベクタと 1 つのスカラにも作用することができます。

ロング命令は、ダブルワード（64 ビット）オペランドを持ち、クワッドワード（128 ワード）の結果を生成します。他のアドバンスト SIMD の乗算命令は、ダブルワードまたはクワッドワードのいずれかのオペランドを持ち、同じサイズの結果を生成することができます。

VFP 乗算命令は、次のものに作用できます。

- 単精度（32 ビット）の浮動小数点数
- 倍精度（64 ビット）の浮動小数点数

一部の VFP 実装は、倍精度数値をサポートしません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.13.6 Miscellaneous Advanced SIMD data-processing instructions

Table A4-21 shows miscellaneous Advanced SIMD data-processing instructions.

Table A4-21 Miscellaneous Advanced SIMD data-processing instructions

Instruction	See
Vector Absolute Difference and Accumulate	*VABA, VABAL* on page A8-526
Vector Absolute Difference	*VABD, VABDL (integer)* on page A8-528 *VABD (floating-point)* on page A8-530
Vector Absolute	*VABS* on page A8-532
Vector Convert between floating-point and fixed point	*VCVT (between floating-point and fixed-point, Advanced SIMD)* on page A8-580
Vector Convert between floating-point and integer	*VCVT (between floating-point and integer, Advanced SIMD)* on page A8-576
Vector Convert between half-precision and single-precision	*VCVT (between half-precision and single-precision, Advanced SIMD)* on page A8-586
Vector Count Leading Sign Bits	*VCLS* on page A8-566
Vector Count Leading Zeros	*VCLZ* on page A8-570
Vector Count Set Bits	*VCNT* on page A8-574
Vector Duplicate scalar	*VDUP (scalar)* on page A8-592
Vector Extract	*VEXT* on page A8-598
Vector Move and Narrow	*VMOVN* on page A8-656
Vector Move Long	*VMOVL* on page A8-654
Vector Maximum, Minimum	*VMAX, VMIN (integer)* on page A8-630 *VMAX, VMIN (floating-point)* on page A8-632
Vector Negate	*VNEG* on page A8-672
Vector Pairwise Maximum, Minimum	*VPMAX, VPMIN (integer)* on page A8-690 *VPMAX, VPMIN (floating-point)* on page A8-692
Vector Reciprocal Estimate	*VRECPE* on page A8-728
Vector Reciprocal Step	*VRECPS* on page A8-730
Vector Reciprocal Square Root Estimate	*VRSQRTE* on page A8-742

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.13.6 アドバンスト SIMD のその他のデータ処理命令

アドバンスト SIMD のその他のデータ処理命令を、表 A4-21 に示します。

表 A4-21 アドバンスト SIMD のその他のデータ処理命令

命令	参照先
ベクタ絶対値の差および累積	*VABA*、*VABAL* P. A8-526
ベクタ絶対値の差	*VABD*、*VABDL（整数）* P. A8-528 *VABD（浮動小数点）* P. A8-530
ベクタ絶対値	*VABS* P. A8-532
浮動小数点と固定小数点とのベクタ変換	*VCVT（浮動小数点数と固定小数点数との間、アドバンスト SIMD）* P. A8-580
浮動小数点と整数とのベクタ変換	*VCVT（浮動小数点数と整数との間、アドバンスト SIMD）* P. A8-576
半精度と単精度とのベクタ変換	*VCVT（半精度と単精度との間、アドバンスト SIMD）* P. A8-586
ベクタ先行符号ビットカウント	*VCLS* P. A8-566
ベクタ先行ゼロカウント	*VCLZ* P. A8-570
ベクタセットビットカウント	*VCNT* P. A8-574
ベクタ複製スカラ	*VDUP（スカラ）* P. A8-592
ベクタ抽出	*VEXT* P. A8-598
ベクタ移動および縮小	*VMOVN* P. A8-656
ベクタロング移動	*VMOVL* P. A8-654
ベクタ最大、最小	*VMAX*、*VMIN（整数）* P. A8-630 *VMAX*、*VMIN（浮動小数点数）* P. A8-632
ベクタ否定	*VNEG* P. A8-672
ベクタペア単位最大、最小	*VPMAX*、*VPMIN（整数）* P. A8-690 *VPMAX*、*VPMIN（浮動小数点数）* P. A8-692
ベクタ逆数概算	*VRECPE* P. A8-728
ベクタ逆数ステップ	*VRECPS* P. A8-730
ベクタ逆数平方根概算	*VRSQRTE* P. A8-742

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A4-21 Miscellaneous Advanced SIMD data-processing instructions (continued)

Instruction	See
Vector Reciprocal Square Root Step	*VRSQRTS* on page A8-744
Vector Reverse	*VREV16, VREV32, VREV64* on page A8-732
Vector Saturating Absolute	*VQABS* on page A8-698
Vector Saturating Move and Narrow	*VQMOVN, VQMOVUN* on page A8-708
Vector Saturating Negate	*VQNEG* on page A8-710
Vector Swap	*VSWP* on page A8-796
Vector Table Lookup	*VTBL, VTBX* on page A8-798
Vector Transpose	*VTRN* on page A8-800
Vector Unzip	*VUZP* on page A8-804
Vector Zip	*VZIP* on page A8-806

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A4-21 アドバンスト SIMD のその他のデータ処理命令（続き）

命令	参照先
ベクタ逆数平方根ステップ	*VRSQRTS* P. A8-744
ベクタ反転	*VREV16*、*VREV32*、*VREV64* P. A8-732
ベクタ飽和絶対値	*VQABS* P. A8-698
ベクタ飽和移動および縮小	*VQMOVN*、*VQMOVUN* P. A8-708
ベクタ飽和否定	*VQNEG* P. A8-710
ベクタスワップ	*VSWP* P. A8-796
ベクタ テーブルルックアップ	*VTBL*、*VTBX* P. A8-798
ベクタ転置	*VTRN* P. A8-800
ベクタアンジップ	*VUZP* P. A8-804
ベクタジップ	*VZIP* P. A8-806

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A4.14 VFP data-processing instructions

Table A4-22 summarizes the data-processing instructions in the VFP instruction set.

For details of the floating-point arithmetic used by VFP instructions, see *Floating-point data types and arithmetic* on page A2-32.

Table A4-22 VFP data-processing instructions

Instruction	See
Absolute value	*VABS* on page A8-532
Add	*VADD (floating-point)* on page A8-538
Compare (optionally with exceptions enabled)	*VCMP, VCMPE* on page A8-572
Convert between floating-point and integer	*VCVT, VCVTR (between floating-point and integer, VFP)* on page A8-578
Convert between floating-point and fixed-point	*VCVT (between floating-point and fixed-point, VFP)* on page A8-582
Convert between double-precision and single-precision	*VCVT (between double-precision and single-precision)* on page A8-584
Convert between half-precision and single-precision	*VCVTB, VCVTT (between half-precision and single-precision, VFP)* on page A8-588
Divide	*VDIV* on page A8-590
Multiply Accumulate, Multiply Subtract	*VMLA, VMLS (floating-point)* on page A8-636
Move immediate value to extension register	*VMOV (immediate)* on page A8-640
Copy from one extension register to another	*VMOV (register)* on page A8-642
Multiply	*VMUL (floating-point)* on page A8-664
Negate (invert the sign bit)	*VNEG* on page A8-672
Multiply Accumulate and Negate, Multiply Subtract and Negate, Multiply and Negate	*VNMLA, VNMLS, VNMUL* on page A8-674
Square Root	*VSQRT* on page A8-762
Subtract	*VSUB (floating-point)* on page A8-790

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A4.14 VFP のデータ処理命令

VFP 命令セットのデータ処理命令の要約を、表 A4-22 に示します。

VFP 命令で使用される浮動小数点演算の詳細については、*浮動小数点データ型と算術演算* P. A2-32 を参照して下さい。

表 A4-22 VFP のデータ処理命令

命令	参照先
絶対値	*VABS* P. A8-532
加算	*VADD（浮動小数点）* P. A8-538
比較（オプションで例外を許可）	*VCMP*、*VCMPE* P. A8-572
浮動小数点と整数との変換	*VCVT*、*VCVTR（浮動小数点数と整数との間、VFP）* P. A8-578
浮動小数点と固定小数点との変換	*VCVT（浮動小数点数と固定小数点数との間、VFP）* P. A8-582
倍精度と単精度との変換	*VCVT（倍精度と単精度との間）* P. A8-584
半精度と単精度との変換	*VCVTB*、*VCVTT（半精度と単精度との間、VFP）* P. A8-588
除算	*VDIV* P. A8-590
積和、積減算	*VMLA*、*VMLS（浮動小数点数）* P. A8-636
イミディエート値を拡張機能レジスタに移動	*VMOV（イミディエート）* P. A8-640
拡張機能レジスタから他の拡張機能レジスタへのコピー	*VMOV（レジスタ）* P. A8-642
乗算	*VMUL*、*VMULL（浮動小数点数）* P. A8-664
否定（符号ビットの反転）	*VNEG* P. A8-672
積和および否定、積減算および否定、乗算および否定	*VNMLA*、*VNMLS*、*VNMUL* P. A8-674
平方根	*VSQRT* P. A8-762
減算	*VSUB（浮動小数点数）* P. A8-790

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter A5
ARM Instruction Set Encoding

This chapter describes the encoding of the ARM instruction set. It contains the following sections:

- *ARM instruction set encoding* on page A5-2
- *Data-processing and miscellaneous instructions* on page A5-4
- *Load/store word and unsigned byte* on page A5-19
- *Media instructions* on page A5-21
- *Branch, branch with link, and block data transfer* on page A5-27
- *Supervisor Call, and coprocessor instructions* on page A5-28
- *Unconditional instructions* on page A5-30.

Note

- Architecture variant information in this chapter describes the architecture variant or extension in which the instruction encoding was introduced into the ARM instruction set. *All* means that the instruction encoding was introduced in ARMv4 or earlier, and so is in all variants of the ARM instruction set covered by this manual.
- In the decode tables in this chapter, an entry of - for a field value means the value of the field does not affect the decoding.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 A5 章
ARM 命令セットのエンコード

本章では、ARM 命令セットのエンコードについて説明します。本章は次のセクションから構成されています。

- *ARM 命令セットのエンコード* P. A5- 2
- *データ処理命令とその他の命令* P. A5- 4
- *ワードおよび符号なしバイトのロード / ストア* P. A5- 19
- *メディア命令* P. A5- 21
- *分岐、リンク付き分岐、ブロックデータ転送* P. A5- 27
- *スーパバイザコールおよびコプロセッサ命令* P. A5- 28
- *無条件命令* P. A5- 30

注

- 本章のアーキテクチャバリアント情報では、その命令エンコードがどのアーキテクチャバリアントや拡張機能で ARM 命令セットに導入されたかについて説明します。すべてとは、その命令エンコードが ARMv4 以前に導入されたもので、本書で扱われている ARM 命令セットのすべてのバリアントに存在することを意味します。
- 本章のデコード表においてフィールド値に「-」が記入されているのは、フィールドの値がデコードに影響を与えないことを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.1 ARM instruction set encoding

31 30 29 28	27 26 25	24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4	3 2 1 0
cond	op1		op	

The ARM instruction stream is a sequence of word-aligned words. Each ARM instruction is a single 32-bit word in that stream.

Table A5-1 shows the major subdivisions of the ARM instruction set, determined by bits [31:25,4].

Most ARM instructions can be conditional, with a condition determined by bits [31:28] of the instruction, the cond field. For details see *The condition field*. This applies to all instructions except those with the cond field equal to 0b1111.

Table A5-1 ARM instruction encoding

cond	op1	op	Instruction classes
not 1111	00x	-	*Data-processing and miscellaneous instructions* on page A5-4.
	010	-	*Load/store word and unsigned byte* on page A5-19.
	011	0	*Load/store word and unsigned byte* on page A5-19.
		1	*Media instructions* on page A5-21.
	10x	-	*Branch, branch with link, and block data transfer* on page A5-27.
	11x	-	*Supervisor Call, and coprocessor instructions* on page A5-28. Includes VFP instructions and Advanced SIMD data transfers, see Chapter A7 *Advanced SIMD and VFP Instruction Encoding*.
1111	-	-	If the cond field is 0b1111, the instruction can only be executed unconditionally, see *Unconditional instructions* on page A5-30. Includes Advanced SIMD instructions, see Chapter A7 *Advanced SIMD and VFP Instruction Encoding*.

A5.1.1 The condition field

Every conditional instruction contains a 4-bit condition code field in bits 31 to 28:

31 30 29 28	27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	

This field contains one of the values 0b0000-0b1110 described in Table A8-1 on page A8-8. Most instruction mnemonics can be extended with the letters defined in the mnemonic extension field.

If the *always* (AL) condition is specified, the instruction is executed irrespective of the value of the condition code flags. The absence of a condition code on an instruction mnemonic implies the AL condition code.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A5.1 ARM 命令セットのエンコード

31 30 29 28	27 26 25	24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4	3 2 1 0
cond	op1		op	

ARM 命令ストリームは、ワードアラインドのワードのシーケンスです。各 ARM 命令は、このストリームに含まれる単一の 32 ビットワードです。

ビット [31:25、4] によって決定される ARM 命令セットの主な分類を、表 A5-1 に示します。

ほとんどの ARM 命令は条件付き実行が可能です。条件は、命令のビット [31:28]、すなわち cond フィールドによって決定されます。詳細については、*条件フィールド*を参照して下さい。これは、cond フィールドが 0b1111 のものを除くすべての命令に該当します。

表 A5-1 ARM 命令のエンコード

cond	op1	op	命令クラス
1111 以外	00x	–	*データ処理命令とその他の命令* P. A5-4
	010	–	*ワードおよび符号なしバイトのロード / ストア* P. A5-19
	011	0	*ワードおよび符号なしバイトのロード / ストア* P. A5-19
		1	*メディア命令* P. A5-21
	10x	–	*分岐、リンク付き分岐、ブロックデータ転送* P. A5-27
	11x	–	*スーパバイザコールおよびコプロセッサ命令* P. A5-28 VFP 命令およびアドバンスト SIMD データ転送を含みます。第 A7 章 *「アドバンスト SIMD および VFP 命令のエンコード」*を参照して下さい。
1111	–	–	cond フィールドが 0b1111 の場合、命令は無条件でしか実行できません。*無条件命令* P. A5-30 を参照して下さい。 アドバンスト SIMD 命令を含みます。第 A7 章 *「アドバンスト SIMD および VFP 命令のエンコード」*を参照して下さい。

A5.1.1 条件フィールド

すべての条件付き命令には、ビット 31 ～ 28 に 4 ビットの条件コードフィールドが含まれます。

31 30 29 28	27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	

このフィールドには、P. A8-8 の表 A8-1 に記載されている 0b0000 ～ 0b1110 のうち 1 つの値が含まれます。ほとんどの命令ニーモニックは、ニーモニック拡張フィールドに定義されている文字で拡張できます。

常時 (AL) 条件が指定されている場合、命令は条件コードフラグの値に関係なく実行されます。命令ニーモニックに条件コードがない場合、それは AL 条件コードであることを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.1.2 UNDEFINED and UNPREDICTABLE instruction set space

An attempt to execute an unallocated instruction results in either:

- Unpredictable behavior. The instruction is described as UNPREDICTABLE.
- An Undefined Instruction exception. The instruction is described as UNDEFINED.

An instruction is UNDEFINED if it is declared as UNDEFINED in an instruction description, or in this chapter.

An instruction is UNPREDICTABLE if:

- it is declared as UNPREDICTABLE in an instruction description or in this chapter
- the pseudocode for that encoding does not indicate that a different special case applies, and a bit marked (0) or (1) in the encoding diagram of an instruction is not 0 or 1 respectively.

Unless otherwise specified:

- ARM instructions introduced in an architecture variant are UNDEFINED in earlier architecture variants.
- ARM instructions introduced in one or more architecture extensions are UNDEFINED if none of those extensions are implemented.

A5.1.3 The PC and the use of 0b1111 as a register specifier

In ARM instructions, the use of 0b1111 as a register specifier specifies the PC.

Many instructions are UNPREDICTABLE if they use 0b1111 as a register specifier. This is specified by pseudocode in the instruction description.

Note

Use of the PC as the base register in any store instruction is deprecated in ARMv7.

A5.1.4 The SP and the use of 0b1101 as a register specifier

In ARM instructions, the use of 0b1101 as a register specifier specifies the SP.

ARM deprecates:

- using SP for any purpose other than as a stack pointer
- using the SP in ARM instructions in ways other that those listed in *32-bit Thumb instruction support for R13* on page A6-4, except that ARM does not deprecate the use of instructions of the following form that write a word-aligned address to SP:

  ```
  SUB SP, <Rd>, #<const>
  ```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.1.2 未定義および予測不能の命令セット空間

未割り当て命令の実行を試みると、次のいずれかの結果が発生します。

- 予測不能な動作。このような命令は、予測不能と記述されています。
- 未定義命令例外。このような命令は、未定義と記述されています。

命令の説明または本章の中で、未定義と記載されている命令は未定義です。

次のいずれかに該当する命令は予測不能です。

- 命令の説明または本章で、予測不能と記載されている。
- 命令のエンコード図で (0) または (1) と表記されたビットがそれぞれ 0 または 1 ではなく、そのエンコードの擬似コードで、別の特別な条件が適用されることが示されていない。

他に特に規定がない場合、次の条件が適用されます。

- あるアーキテクチャバリアントで導入された ARM 命令は、それ以前のアーキテクチャバリアントでは未定義です。
- 1 つ以上のアーキテクチャ拡張機能に導入されている ARM 命令は、その拡張機能が実装されていない場合は未定義です。

A5.1.3 レジスタ指定子としての PC および 0b1111 の使用

ARM 命令では、レジスタ指定子に 0b1111 を使用すると PC が指定されます。

多くの命令は、レジスタ指定子に 0b1111 を使用している場合、予測不能です。これは、命令説明の擬似コードによって指定されています。

注

ストア命令で PC をベースレジスタとして使用することは、ARMv7 では推奨されません。

A5.1.4 レジスタ指定子としての SP および 0b1101 の使用

ARM 命令では、レジスタ指定子に 0b1101 を使用すると SP が指定されます。

次の操作は推奨されません。

- スタックポインタ以外の目的で SP を使用すること。
- *32 ビット Thumb 命令での R13 のサポート* P. A6- 4 に列記されている以外の形で ARM 命令に SP を使用すること。ただし、SP にワードアラインド アドレスを書き込む次の形の命令を使用することは問題ありません。

  ```
  SUB SP, <Rd>, #<const>
  ```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2 Data-processing and miscellaneous instructions

31 30 29 28	27 26	25	24 23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0	op	op1		op2	

Table A5-2 shows the allocation of encodings in this space.

Table A5-2 Data-processing and miscellaneous instructions

op	op1	op2	Instruction or instruction class	Variant
0	not 10xx0	xxx0	*Data-processing (register)* on page A5-5	-
		0xx1	*Data-processing (register-shifted register)* on page A5-7	-
	10xx0	0xxx	*Miscellaneous instructions* on page A5-18	-
		1xx0	*Halfword multiply and multiply-accumulate* on page A5-13	-
	0xxxx	1001	*Multiply and multiply-accumulate* on page A5-12	-
	1xxxx	1001	*Synchronization primitives* on page A5-16	-
	not 0xx1x	1011	*Extra load/store instructions* on page A5-14	-
		11x1	*Extra load/store instructions* on page A5-14	-
	0xx1x	1011	*Extra load/store instructions (unprivileged)* on page A5-15	-
		11x1	*Extra load/store instructions (unprivileged)* on page A5-15	-
1	not 10xx0	-	*Data-processing (immediate)* on page A5-8	-
	10000	-	16-bit immediate load (*MOV (immediate)* on page A8-194)	v6T2
	10100	-	High halfword 16-bit immediate load (*MOVT* on page A8-200)	v6T2
	10x10	-	*MSR (immediate), and hints* on page A5-17	-

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2 データ処理命令とその他の命令

31 30 29 28	27 26	25	24 23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0	op	op1		op2	

この空間でのエンコードの割り当てを、表 A5-2 に示します。

表 A5-2 データ処理命令とその他の命令

op	op1	op2	命令または命令クラス	バリアント
0	10xx0 以外	xxx0	*データ処理（レジスタ）* P. A5-5	–
		0xx1	*データ処理（レジスタによりシフトしたレジスタ）* P. A5-7	–
	10xx0	0xxx	*その他の命令* P. A5-18	–
		1xx0	*ハーフワード乗算および積和* P. A5-13	–
	0xxxx	1001	*乗算および積和演算* P. A5-12	–
	1xxxx	1001	*同期化基本命令* P. A5-16	–
	0xx1x 以外	1011	*追加のロード／ストア命令* P. A5-14	–
		11x1	*追加のロード／ストア命令* P. A5-14	–
	0xx1x	1011	*追加のロード／ストア命令（非特権）* P. A5-15	–
		11x1	*追加のロード／ストア命令（非特権）* P. A5-15	–
1	10xx0 以外	–	*データ処理（イミディエート）* P. A5-8	–
	10000	–	16 ビットのイミディエートロード（*MOV（イミディエート）* P. A8-194）	v6T2
	10100	–	上位ハーフワード 16 ビットのイミディエートロード（*MOVT* P. A8-200）	v6T2
	10x10	–	*MSR（イミディエート）およびヒント* P. A5-17	–

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.1 Data-processing (register)

31 30 29 28	27 26 25	24 23 22 21	20 19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0 0	op1			op2		op3	0	

If op1 == 0b10xx0, see *Data-processing and miscellaneous instructions* on page A5-4.

Table A5-3 shows the allocation of encodings in this space. These encodings are in all architecture variants.

Table A5-3 Data-processing (register) instructions

op1	op2	op3	Instruction	See
0000x	-	-	Bitwise AND	*AND (register)* on page A8-36
0001x	-	-	Bitwise Exclusive OR	*EOR (register)* on page A8-96
0010x	-	-	Subtract	*SUB (register)* on page A8-422
0011x	-	-	Reverse Subtract	*RSB (register)* on page A8-286
0100x	-	-	Add	*ADD (register)* on page A8-24
0101x	-	-	Add with Carry	*ADC (register)* on page A8-16
0110x	-	-	Subtract with Carry	*SBC (register)* on page A8-304
0111x	-	-	Reverse Subtract with Carry	*RSC (register)* on page A8-292
10001	-	-	Test	*TST (register)* on page A8-456
10011	-	-	Test Equivalence	*TEQ (register)* on page A8-450
10101	-	-	Compare	*CMP (register)* on page A8-82
10111	-	-	Compare Negative	*CMN (register)* on page A8-76
1100x	-	-	Bitwise OR	*ORR (register)* on page A8-230
1101x	00000	00	Move	*MOV (register)* on page A8-196
	not 00000	00	Logical Shift Left	*LSL (immediate)* on page A8-178
	-	01	Logical Shift Right	*LSR (immediate)* on page A8-182
	-	10	Arithmetic Shift Right	*ASR (immediate)* on page A8-40
	00000	11	Rotate Right with Extend	*RRX* on page A8-282
	not 00000	11	Rotate Right	*ROR (immediate)* on page A8-278

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.1 データ処理（レジスタ）

31 30 29 28	27 26 25	24 23 22 21	20 19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0 0	op1			op2		op3	0	

op1 == 0b10xx0 の場合は、*データ処理命令とその他の命令* P. A5-4 を参照して下さい。

この空間でのエンコードの割り当てを、表 A5-3 に示します。これらのエンコードは、すべてのアーキテクチャバリアントに存在します。

表 A5-3 データ処理（レジスタ）命令

op1	op2	op3	命令	参照先
0000x	–	–	ビット単位論理積	*AND（レジスタ）* P. A8-36
0001x	–	–	ビット単位排他的論理和	*EOR（レジスタ）* P. A8-96
0010x	–	–	減算	*SUB（レジスタ）* P. A8-422
0011x	–	–	逆減算	*RSB（レジスタ）* P. A8-286
0100x	–	–	加算	*ADD（レジスタ）* P. A8-24
0101x	–	–	キャリー付き加算	*ADC（レジスタ）* P. A8-16
0110x	–	–	キャリー付き減算	*SBC（レジスタ）* P. A8-304
0111x	–	–	キャリー付き逆減算	*RSC（レジスタ）* P. A8-292
10001	–	–	テスト	*TST（レジスタ）* P. A8-456
10011	–	–	等価テスト	*TEQ（レジスタ）* P. A8-450
10101	–	–	比較	*CMP（レジスタ）* P. A8-82
10111	–	–	比較否定	*CMN（レジスタ）* P. A8-76
1100x	–	–	ビット単位論理和	*ORR（レジスタ）* P. A8-230
1101x	00000	00	移動	*MOV（レジスタ）* P. A8-196
	00000 以外	00	論理左シフト	*LSL（イミディエート）* P. A8-178
	–	01	論理右シフト	*LSR（イミディエート）* P. A8-182
	–	10	算術右シフト	*ASR（イミディエート）* P. A8-40
	00000	11	拡張付き右ローテート	*RRX* P. A8-282
	00000 以外	11	右ローテート	*ROR（イミディエート）* P. A8-278

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A5-3 Data-processing (register) instructions (continued)

op1	op2	op3	Instruction	See
1110x	-	-	Bitwise Bit Clear	*BIC (register)* on page A8-52
1111x	-	-	Bitwise NOT	*MVN (register)* on page A8-216

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A5-3 データ処理（レジスタ）命令（続き）

op1	op2	op3	命令	参照先
1110x	–	–	ビット単位のビットクリア	*BIC（レジスタ）* P. A8-52
1111x	–	–	ビット単位否定	*MVN（レジスタ）* P. A8-216

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.2 Data-processing (register-shifted register)

31 30 29 28	27 26 25	24 23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0 0	op1		0	op2	1	

If op1 == 0b10xx0, see *Data-processing and miscellaneous instructions* on page A5-4.

Table A5-4 shows the allocation of encodings in this space. These encodings are in all architecture variants.

Table A5-4 Data-processing (register-shifted register) instructions

op1	op2	Instruction	See
0000x	-	Bitwise AND	*AND (register-shifted register)* on page A8-38
0001x	-	Bitwise Exclusive OR	*EOR (register-shifted register)* on page A8-98
0010x	-	Subtract	*SUB (register-shifted register)* on page A8-424
0011x	-	Reverse Subtract	*RSB (register-shifted register)* on page A8-288
0100x	-	Add	*ADD (register-shifted register)* on page A8-26
0101x	-	Add with Carry	*ADC (register-shifted register)* on page A8-18
0110x	-	Subtract with Carry	*SBC (register-shifted register)* on page A8-306
0111x	-	Reverse Subtract with Carry	*RSC (register-shifted register)* on page A8-294
10001	-	Test	*TST (register-shifted register)* on page A8-458
10011	-	Test Equivalence	*TEQ (register-shifted register)* on page A8-452
10101	-	Compare	*CMP (register-shifted register)* on page A8-84
10111	-	Compare Negative	*CMN (register-shifted register)* on page A8-78
1100x	-	Bitwise OR	*ORR (register-shifted register)* on page A8-232
1101x	00	Logical Shift Left	*LSL (register)* on page A8-180
	01	Logical Shift Right	*LSR (register)* on page A8-184
	10	Arithmetic Shift Right	*ASR (register)* on page A8-42
	11	Rotate Right	*ROR (register)* on page A8-280
1110x	-	Bitwise Bit Clear	*BIC (register-shifted register)* on page A8-54
1111x	-	Bitwise NOT	*MVN (register-shifted register)* on page A8-218

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.2 データ処理（レジスタによりシフトしたレジスタ）

31 30 29 28	27 26 25	24 23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0 0	op1		0	op2	1	

op1 == 0b10xx0 の場合は、*データ処理命令とその他の命令* P. A5-4 を参照して下さい。

この空間でのエンコードの割り当てを、表 A5-4 に示します。これらのエンコードは、すべてのアーキテクチャバリアントに存在します。

表 A5-4 データ処理（レジスタによりシフトしたレジスタ）命令

op1	op2	命令	参照先
0000x	–	ビット単位論理積	*AND（レジスタによりシフトしたレジスタ）* P. A8-38
0001x	–	ビット単位排他的論理和	*EOR（レジスタによりシフトしたレジスタ）* P. A8-98
0010x	–	減算	*SUB（レジスタによりシフトしたレジスタ）* P. A8-424
0011x	–	逆減算	*RSB（レジスタによりシフトしたレジスタ）* P. A8-288
0100x	–	加算	*ADD（レジスタによりシフトしたレジスタ）* P. A8-26
0101x	–	キャリー付き加算	*ADC（レジスタよりシフトしたレジスタ）* P. A8-18
0110x	–	キャリー付き減算	*SBC（レジスタによりシフトしたレジスタ）* P. A8-306
0111x	–	キャリー付き逆減算	*RSC（レジスタによりシフトしたレジスタ）* P. A8-294
10001	–	テスト	*TST（レジスタによりシフトしたレジスタ）* P. A8-458
10011	–	等価テスト	*TEQ（レジスタによりシフトしたレジスタ）* P. A8-452
10101	–	比較	*CMP（レジスタによりシフトしたレジスタ）* P. A8-84
10111	–	比較否定	*CMN（レジスタによりシフトしたレジスタ）* P. A8-78
1100x	–	ビット単位論理和	*ORR（レジスタによりシフトしたレジスタ）* P. A8-232
1101x	00	論理左シフト	*LSL（レジスタ）* P. A8-180
	01	論理右シフト	*LSR（レジスタ）* P. A8-184
	10	算術右シフト	*ASR（レジスタ）* P. A8-42
	11	右ローテート	*ROR（レジスタ）* P. A8-280
1110x	–	ビット単位のビットクリア	*BIC（レジスタによりシフトしたレジスタ）* P. A8-54
1111x	–	ビット単位否定	*MVN（レジスタによりシフトしたレジスタ）* P. A8-218

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.3 Data-processing (immediate)

31 30 29 28	27 26 25	24 23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0 1	op	Rn	

If op == 0b10xx0, see *Data-processing and miscellaneous instructions* on page A5-4.

Table A5-5 shows the allocation of encodings in this space. These encodings are in all architecture variants.

Table A5-5 Data-processing (immediate) instructions

op	Rn	Instruction	See
0000x	-	Bitwise AND	*AND (immediate)* on page A8-34
0001x	-	Bitwise Exclusive OR	*EOR (immediate)* on page A8-94
0010x	not 1111	Subtract	*SUB (immediate, ARM)* on page A8-420
	1111	Form PC-relative address	*ADR* on page A8-32
0011x	-	Reverse Subtract	*RSB (immediate)* on page A8-284
0100x	not 1111	Add	*ADD (immediate, ARM)* on page A8-22
	1111	Form PC-relative address	*ADR* on page A8-32
0101x	-	Add with Carry	*ADC (immediate)* on page A8-14
0110x	-	Subtract with Carry	*SBC (immediate)* on page A8-302
0111x	-	Reverse Subtract with Carry	*RSC (immediate)* on page A8-290
10001	-	Test	*TST (immediate)* on page A8-454
10011	-	Test Equivalence	*TEQ (immediate)* on page A8-448
10101	-	Compare	*CMP (immediate)* on page A8-80
10111	-	Compare Negative	*CMN (immediate)* on page A8-74
1100x	-	Bitwise OR	*ORR (immediate)* on page A8-228
1101x	-	Move	*MOV (immediate)* on page A8-194
1110x	-	Bitwise Bit Clear	*BIC (immediate)* on page A8-50
1111x	-	Bitwise NOT	*MVN (immediate)* on page A8-214

These instructions all have modified immediate constants, rather than a simple 12-bit binary number. This provides a more useful range of values. For details see *Modified immediate constants in ARM instructions* on page A5-9.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A5.2.3 データ処理（イミディエート）

31 30 29 28	27 26 25	24 23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0 1	op	Rn	

op == 0b10xx0 の場合は、*データ処理命令とその他の命令* P. A5-4 を参照して下さい。

この空間でのエンコードの割り当てを、表 A5-5 に示します。これらのエンコードは、すべてのアーキテクチャバリアントに存在します。

表 A5-5 データ処理（イミディエート）命令

op	Rn	命令	参照先
0000x	–	ビット単位論理積	*AND（イミディエート）* P. A8-34
0001x	–	ビット単位排他的論理和	*EOR（イミディエート）* P. A8-94
0010x	1111 以外	減算	*SUB（イミディエート、ARM）* P. A8-420
	1111	PC 相対アドレスの生成	*ADR* P. A8-32
0011x	–	逆減算	*RSB（イミディエート）* P. A8-284
0100x	1111 以外	加算	*ADD（イミディエート、ARM）* P. A8-22
	1111	PC 相対アドレスの生成	*ADR* P. A8-32
0101x	–	キャリー付き加算	*ADC（イミディエート）* P. A8-14
0110x	–	キャリー付き減算	*SBC（イミディエート）* P. A8-302
0111x	–	キャリー付き逆減算	*RSC（イミディエート）* P. A8-290
10001	–	テスト	*TST（イミディエート）* P. A8-454
10011	–	等価テスト	*TEQ（イミディエート）* P. A8-448
10101	–	比較	*CMP（イミディエート）* P. A8-80
10111	–	比較否定	*CMN（イミディエート）* P. A8-74
1100x	–	ビット単位論理和	*ORR（イミディエート）* P. A8-228
1101x	–	移動	*MOV（イミディエート）* P. A8-194
1110x	–	ビット単位のビットクリア	*BIC（イミディエート）* P. A8-50
1111x	–	ビット単位否定	*MVN（イミディエート）* P. A8-214

これらの命令はすべて、単純な 12 ビットバイナリ数値ではなく、修飾イミディエート定数を使用します。これによって、値の範囲がより広く応用可能です。詳細については、*ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.4 Modified immediate constants in ARM instructions

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
rotation	a b c d e f g h

Table A5-6 shows the range of modified immediate constants available in ARM data-processing instructions, and how they are encoded in the a, b, c, d, e, f, g, h, and rotation fields in the instruction.

Table A5-6 Encoding of modified immediates in ARM processing instructions

rotation	<const> [a]
0000	00000000 00000000 00000000 abcdefgh
0001	gh000000 00000000 00000000 00abcdef
0010	efgh0000 00000000 00000000 0000abcd
0011	cdefgh00 00000000 00000000 000000ab
0100	abcdefgh 00000000 00000000 00000000
.	.
.	. 8-bit values shifted to other even-numbered positions
.	.
1001	00000000 00abcdef gh000000 00000000
.	.
.	. 8-bit values shifted to other even-numbered positions
.	.
1110	00000000 00000000 0000abcd efgh0000
1111	00000000 00000000 000000ab cdefgh00

a. In this table, the immediate constant value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

Note

The range of values available in ARM modified immediate constants is slightly different from the range of values available in 32-bit Thumb instructions. See *Modified immediate constants in Thumb instructions* on page A6-17.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A5.2.4 ARM 命令での修飾イミディエート定数

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
rotation	a b c d e f g h

ARM データ処理命令で使用可能な修飾イミディエート定数の範囲と、それらがこの命令内の a、b、c、d、e、f、g、h の各フィールドと rotation フィールドにどのようにエンコードされるかを、表 A5-6 に示します。

表 A5-6 ARM 処理命令での修飾イミディエート値のエンコード

rotation	＜定数＞ [a]
0000	00000000 00000000 00000000 abcdefgh
0001	gh000000 00000000 00000000 00abcdef
0010	efgh0000 00000000 00000000 0000abcd
0011	cdefgh00 00000000 00000000 000000ab
0100	abcdefgh 00000000 00000000 00000000
.	.
.	. 他の偶数番号の位置にシフトされた 8 ビット値
.	.
1001	00000000 00abcdef gh000000 00000000
.	.
.	. 他の偶数番号の位置にシフトされた 8 ビット値
.	.
1110	00000000 00000000 0000abcd efgh0000
1111	00000000 00000000 000000ab cdefgh00

a. この表では、abcdefgh をエンコード図に関連付けるため、イミディエート定数値をバイナリ形式で示しています。アセンブラ構文では、イミディエート値を通常の方法（デフォルトでは 10 進数）で指定します。

注

ARM 修飾イミディエート定数に使用できる値の範囲は、32 ビット Thumb 命令で使用可能な値の範囲とはやや異なります。詳細については、*Thumb 命令での修飾イミディエート定数* P. A6-17 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Carry out

A logical instruction with rotation == 0b0000 does not affect APSR.C. Otherwise, a logical instruction that sets the flags sets APSR.C to the value of bit [31] of the modified immediate constant.

Constants with multiple encodings

Some constant values have multiple possible encodings. In this case, a UAL assembler must select the encoding with the lowest unsigned value of the rotation field. This is the encoding that appears first in Table A5-6 on page A5-9. For example, the constant #3 must be encoded with (rotation, abcdefgh) == (0b0000, 0b00000011), not (0b0001, 0b00001100), (0b0010, 0b00110000), or (0b0011, 0b11000000).

In particular, this means that all constants in the range 0-255 are encoded with rotation == 0b0000, and permitted constants outside that range are encoded with rotation != 0b0000. A flag-setting logical instruction with a modified immediate constant therefore leaves APSR.C unchanged if the constant is in the range 0-255 and sets it to the most significant bit of the constant otherwise. This matches the behavior of Thumb modified immediate constants for all constants that are permitted in both the ARM and Thumb instruction sets.

An alternative syntax is available for a modified immediate constant that permits the programmer to specify the encoding directly. In this syntax, #<const> is instead written as #<byte>,#<rot>, where:

<byte> is the numeric value of abcdefgh, in the range 0-255

<rot> is twice the numeric value of rotation, an even number in the range 0-30.

This syntax permits all ARM data-processing instructions with modified immediate constants to be disassembled to assembler syntax that will assemble to the original instruction.

This syntax also makes it possible to write variants of some flag-setting logical instructions that have different effects on APSR.C to those obtained with the normal #<const> syntax. For example, ANDS R1,R2,#12,#2 has the same behavior as ANDS R1,R2,#3 except that it sets APSR.C to 0 instead of leaving it unchanged. Such variants of flag-setting logical instructions do not have equivalents in the Thumb instruction set, and their use is deprecated.

Operation

```
// ARMExpandImm()
// ==============

bits(32) ARMExpandImm(bits(12) imm12)

    // APSR.C argument to following function call does not affect the imm32 result.
    (imm32, -) = ARMExpandImm_C(imm12, APSR.C);

    return imm32;

// ARMExpandImm_C()
// ================

(bits(32), bit) ARMExpandImm_C(bits(12) imm12, bit carry_in)
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

キャリーアウト

ローテート == 0b0000 の論理命令は APSR.C に影響を与えません。それ以外の場合、フラグをセットする論理命令によって、APSR.C が修飾イミディエート定数のビット [31] の値に設定されます。

複数のエンコードを持つ定数

一部の定数は、複数のエンコードが可能です。この場合、UAL アセンブラは rotation フィールドの符号なしの値が最も小さくなるエンコードを選択しなければなりません。これは P. A5-9 の表 A5-6 の最初に示されているエンコードです。例えば、定数 #3 は、(rotation, abcdefgh) == (0b0001, 0b00001100)、(0b0010, 0b00110000)、(0b0011, 0b11000000) ではなく、(rotation, abcdefgh) == (0b0000, 0b00000011) でエンコードする必要があります。

つまり、0 ～ 255 の範囲にあるすべての定数は rotation == 0b0000 でエンコードされます。その範囲外で許可される定数は、rotation != 0b0000 でエンコードされます。したがって、定数が 0 ～ 255 の範囲にある場合、修飾イミディエート定数を持ち、フラグをセットする論理命令は、APSR.C を変更しません。そうでない場合は、APSR.C を定数の最上位ビットにセットします。これは、ARM 命令セットと Thumb 命令セットの両方で許可されているすべての定数について、Thumb 修飾イミディエート定数の動作に一致します。

プログラマがエンコードを直接指定できる修飾イミディエート定数については、代替構文が使用できます。この構文では、次の場合、#<const> を #<byte>,#<rot> と記述します。

<byte> が、abcdefgh の数値で、0 ～ 255 の範囲にある。

<rot> が、rotation の数値の 2 倍で、0 ～ 30 の範囲の偶数である。

この構文では、修飾イミディエート定数を持つすべての ARM データ処理命令を、元の命令にアセンブルされるアセンブラ構文に逆アセンブルすることができます。

また、フラグをセットする一部の論理命令のバリアントを記述し、通常の #<const> 構文で得られるのとは異なる効果を APSR.C に与えることもできます。例えば、ANDS R1,R2,#12,#2 は、APSR.C を変更しないのではなく、0 にセットするという点以外、ANDS R1,R2,#3 と同じ動作をします。このようなフラグをセットする論理命令のバリアントは、Thumb 命令セットには同等の命令が存在せず、使用は推奨されません。

動作


```
// ARMExpandImm()
// ==============

bits(32) ARMExpandImm(bits(12) imm12)

    // APSR.C argument to following function call does not affect the imm32 result.
    (imm32, -) = ARMExpandImm_C(imm12, APSR.C);

    return imm32;

// ARMExpandImm_C()
// ================
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
unrotated_value = ZeroExtend(imm12<7:0>, 32);
(imm32, carry_out) = Shift_C(unrotated_value, SRType_ROR, 2*UInt(imm12<11:8>), carry_in);

return (imm32, carry_out);
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
(bits(32), bit) ARMExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRType_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.5 Multiply and multiply-accumulate

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0	op		1 0 0 1	

Table A5-7 shows the allocation of encodings in this space.

Table A5-7 Multiply and multiply-accumulate instructions

op	Instruction	See	Variant
000x	Multiply	*MUL* on page A8-212	All
001x	Multiply Accumulate	*MLA* on page A8-190	All
0100	Unsigned Multiply Accumulate Accumulate Long	*UMAAL* on page A8-482	v6
0101	UNDEFINED	-	-
0110	Multiply and Subtract	*MLS* on page A8-192	v6T2
0111	UNDEFINED	-	-
100x	Unsigned Multiply Long	*UMULL* on page A8-486	All
101x	Unsigned Multiply Accumulate Long	*UMLAL* on page A8-484	All
110x	Signed Multiply Long	*SMULL* on page A8-356	All
111x	Signed Multiply Accumulate Long	*SMLAL* on page A8-334	All

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A5.2.5 乗算および積和演算

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0	op		1 0 0 1	

この空間でのエンコードの割り当てを、表 A5-7 に示します。

表 A5-7 乗算および積和演算命令

op	命令	参照先	バリアント
000x	乗算	*MUL* P. A8-212	すべて
001x	積和演算	*MLA* P. A8-190	すべて
0100	符号なしロング積和和演算	*UMAAL* P. A8-482	v6
0101	未定義	–	–
0110	積減算	*MLS* P. A8-192	v6T2
0111	未定義	–	–
100x	符号なしロング乗算	*UMULL* P. A8-486	すべて
101x	符号なしロング積和演算	*UMLAL* P. A8-484	すべて
110x	符号付きロング乗算	*SMULL* P. A8-356	すべて
111x	符号付きロング積和演算	*SMLAL* P. A8-334	すべて

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.6 Saturating addition and subtraction

31 30 29 28	27 26 25 24	23	22 21	20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	0	op	0		0 1 0 1	

Table A5-8 shows the allocation of encodings in this space. These encodings are all available in ARMv5TE and above, and are UNDEFINED in earlier variants of the architecture.

Table A5-8 Saturating addition and subtraction instructions

op	Instruction	See
00	Saturating Add	*QADD* on page A8-250
01	Saturating Subtract	*QSUB* on page A8-264
10	Saturating Double and Add	*QDADD* on page A8-258
11	Saturating Double and Subtract	*QDSUB* on page A8-260

A5.2.7 Halfword multiply and multiply-accumulate

31 30 29 28	27 26 25 24	23	22 21	20	19 18 17 16 15 14 13 12 11 10 9 8	7	6	5	4	3 2 1 0
cond	0 0 0 1	0	op1	0		1		op	0	

Table A5-9 shows the allocation of encodings in this space.

These encodings are signed multiply (SMUL) and signed multiply-accumulate (SMLA) instructions, operating on 16-bit values, or mixed 16-bit and 32-bit values. The results and accumulators are 32-bit or 64-bit.

These encodings are all available in ARMv5TE and above, and are UNDEFINED in earlier variants of the architecture.

Table A5-9 Halfword multiply and multiply-accumulate instructions

op1	op	Instruction	See
00	-	Signed 16-bit multiply, 32-bit accumulate	*SMLABB, SMLABT, SMLATB, SMLATT* on page A8-330
01	0	Signed 16-bit x 32-bit multiply, 32-bit accumulate	*SMLAWB, SMLAWT* on page A8-340
01	1	Signed 16-bit x 32-bit multiply, 32-bit result	*SMULWB, SMULWT* on page A8-358
10	-	Signed 16-bit multiply, 64-bit accumulate	*SMLALBB, SMLALBT, SMLALTB, SMLALTT* on page A8-336
11	-	Signed 16-bit multiply, 32-bit result	*SMULBB, SMULBT, SMULTB, SMULTT* on page A8-354

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.6 飽和加算および減算

31 30 29 28	27 26 25 24	23	22 21	20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	0	op	0		0 1 0 1	

この空間でのエンコードの割り当てを、表 A5-8 に示します。これらのエンコードはすべて、ARMv5TE およびそれ以降で使用できます。それ以前のアーキテクチャバリアントでは未定義です。

表 A5-8 飽和加算および減算命令

op	命令	参照先
00	飽和加算	*QADD* P. A8-250
01	飽和減算	*QSUB* P. A8-264
10	ダブル飽和加算	*QDADD* P. A8-258
11	ダブル飽和減算	*QDSUB* P. A8-260

A5.2.7 ハーフワード乗算および積和

31 30 29 28	27 26 25 24	23	22 21	20	19 18 17 16 15 14 13 12 11 10 9 8	7	6	5	4	3 2 1 0
cond	0 0 0 1	0	op1	0		1		op	0	

この空間でのエンコードの割り当てを、表 A5-9 に示します。

これらのエンコードは、16 ビット値、あるいは 16 ビット値と 32 ビット値の混合に作用する符号付き乗算 (SMUL) と符号付き積和 (SMLA) 命令です。結果と累算器は 32 ビットまたは 64 ビットです。

これらのエンコードはすべて、ARMv5TE およびそれ以降で使用できます。それ以前のアーキテクチャバリアントでは未定義です。

表 A5-9 ハーフワード乗算および積和命令

op1	op	命令	参照先
00	–	符号付き 16 ビット乗算、32 ビット累積	*SMLABB*、*SMLABT*、*SMLATB*、*SMLATT* P. A8-330
01	0	符号付き 16 ビット × 32 ビット乗算、32 ビット累積	*SMLAWB*、*SMLAWT* P. A8-340
01	1	符号付き 16 ビット × 32 ビット乗算、32 ビット結果	*SMULWB*、*SMULWT* P. A8-358
10	–	符号付き 16 ビット乗算、64 ビット累積	*SMLALBB*、*SMLALBT*、*SMLALTB*、*SMLALTT* P. A8-336
11	–	符号付き 16 ビット乗算、32 ビット結果	*SMULBB*、*SMULBT*、*SMULTB*、*SMULTT* P. A8-354

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.8 Extra load/store instructions

31 30 29 28	27 26 25	24 23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0 0	op1	Rn		1	op2	1	

If op1 == 0b0xx1x or op2 == 0b00, see *Data-processing and miscellaneous instructions* on page A5-4.

Table A5-10 shows the allocation of encodings in this space.

Table A5-10 Extra load/store instructions

op2	op1	Rn	Instruction	See	Variant
01	xx0x0	-	Store Halfword	*STRH (register)* on page A8-412	All
	xx0x1	-	Load Halfword	*LDRH (register)* on page A8-156	All
	xx1x0	-	Store Halfword	*STRH (immediate, ARM)* on page A8-410	All
	xx1x1	not 1111	Load Halfword	*LDRH (immediate, ARM)* on page A8-152	All
		1111	Load Halfword	*LDRH (literal)* on page A8-154	All
10	xx0x0	-	Load Dual	*LDRD (register)* on page A8-140	v5TE
	xx0x1	-	Load Signed Byte	*LDRSB (register)* on page A8-164	All
	xx1x0	not 1111	Load Dual	*LDRD (immediate)* on page A8-136	v5TE
		1111	Load Dual	*LDRD (literal)* on page A8-138	v5TE
	xx1x1	not 1111	Load Signed Byte	*LDRSB (immediate)* on page A8-160	All
		1111	Load Signed Byte	*LDRSB (literal)* on page A8-162	All
11	xx0x0	-	Store Dual	*STRD (register)* on page A8-398	All
	xx0x1	-	Load Signed Halfword	*LDRSH (register)* on page A8-172	All
	xx1x0	-	Store Dual	*STRD (immediate)* on page A8-396	All
	xx1x1	not 1111	Load Signed Halfword	*LDRSH (immediate)* on page A8-168	All
		1111	Load Signed Halfword	*LDRSH (literal)* on page A8-170	All

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.8 追加のロード / ストア命令

31 30 29 28	27 26 25	24 23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0 0	op1	Rn		1	op2	1	

op1 == 0b0xx1x または op2 == 0b00 の場合は、*データ処理命令とその他の命令* P. A5-4 を参照して下さい。

この空間でのエンコードの割り当てを、表 A5-10 に示します。

表 A5-10 追加のロード / ストア命令

op2	op1	Rn	命令	参照先	バリアント
01	xx0x0	–	ハーフワードストア	*STRH（レジスタ）* P. A8-412	すべて
	xx0x1	–	ハーフワードロード	*LDRH（レジスタ）* P. A8-156	すべて
	xx1x0	–	ハーフワードストア	*STRH（イミディエート、ARM）* P. A8-410	すべて
	xx1x1	1111 以外	ハーフワードロード	*LDRH（イミディエート、ARM）* P. A8-152	すべて
		1111	ハーフワードロード	*LDRH（リテラル）* P. A8-154	すべて
10	xx0x0	–	デュアルロード	*LDRD（レジスタ）* P. A8-140	v5TE
	xx0x1	–	符号付きバイトロード	*LDRSB（レジスタ）* P. A8-164	すべて
	xx1x0	1111 以外	デュアルロード	*LDRD（イミディエート）* P. A8-136	v5TE
		1111	デュアルロード	*LDRD（リテラル）* P. A8-138	v5TE
	xx1x1	1111 以外	符号付きバイトロード	*LDRSB（イミディエート）* P. A8-160	すべて
		1111	符号付きバイトロード	*LDRSB（リテラル）* P. A8-162	すべて
11	xx0x0	–	デュアルストア	*STRD（レジスタ）* P. A8-398	すべて
	xx0x1	–	符号付きハーフワードロード	*LDRSH（レジスタ）* P. A8-172	すべて
	xx1x0	–	デュアルストア	*STRD（イミディエート）* P. A8-396	すべて
	xx1x1	1111 以外	符号付きハーフワードロード	*LDRSH（イミディエート）* P. A8-168	すべて
		1111	符号付きハーフワードロード	*LDRSH（リテラル）* P. A8-170	すべて

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.9 Extra load/store instructions (unprivileged)

31 30 29 28	27 26 25 24	23 22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0 0 0		1	op		Rt		1	op2	1	

If op2 == 0b00, see *Data-processing and miscellaneous instructions* on page A5-4.

Table A5-11 shows the allocation of encodings in this space. The instruction encodings are all available in ARMv6T2 and above, and are UNDEFINED in earlier variants of the architecture.

Table A5-11 Extra load/store instructions (unprivileged)

op2	op	Rt	Instruction	See
01	0	-	Store Halfword Unprivileged	*STRHT* on page A8-414
	1	-	Load Halfword Unprivileged	*LDRHT* on page A8-158
1x	0	xxx0	UNPREDICTABLE	-
		xxx1	UNDEFINED	-
10	1	-	Load Signed Byte Unprivileged	*LDRSBT* on page A8-166
11	1	-	Load Signed Halfword Unprivileged	*LDRSHT* on page A8-174

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.9 追加のロード / ストア命令（非特権）

31 30 29 28	27 26 25 24	23 22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0 0 0		1	op		Rt		1	op2	1	

op2 == 0b00 の場合は、*データ処理命令とその他の命令* P. A5-4 を参照して下さい。

この空間でのエンコードの割り当てを、表 A5-11 に示します。命令のエンコードはすべて、ARMv6T2 およびそれ以降で使用できます。それ以前のアーキテクチャバリアントでは未定義です。

表 A5-11 追加のロード / ストア命令（非特権）

op2	op	Rt	命令	参照先
01	0	–	非特権ハーフワードストア	*STRHT* P. A8-414
	1	–	非特権ハーフワードロード	*LDRHT* P. A8-158
1x	0	xxx0	予測不能	–
		xxx1	未定義	–
10	1	–	非特権符号付きバイトロード	*LDRSBT* P. A8-166
11	1	–	非特権符号付きハーフワードロード	*LDRSHT* P. A8-174

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.10 Synchronization primitives

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	op		1 0 0 1	

Table A5-12 shows the allocation of encodings in this space.

Other encodings in this space are UNDEFINED.

Table A5-12 Synchronization primitives

op	Instruction	See	Variant
0x00	Swap Word, Swap Byte	*SWP, SWPB* on page A8-432 [a]	All
1000	Store Register Exclusive	*STREX* on page A8-400	v6
1001	Load Register Exclusive	*LDREX* on page A8-142	v6
1010	Store Register Exclusive Doubleword	*STREXD* on page A8-404	v6K
1011	Load Register Exclusive Doubleword	*LDREXD* on page A8-146	v6K
1100	Store Register Exclusive Byte	*STREXB* on page A8-402	v6K
1101	Load Register Exclusive Byte	*LDREXB* on page A8-144	v6K
1110	Store Register Exclusive Halfword	*STREXH* on page A8-406	v6K
1111	Load Register Exclusive Halfword	*LDREXH* on page A8-148	v6K

a. Use of these instructions is deprecated.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.10 同期化基本命令

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	op		1 0 0 1	

この空間でのエンコードの割り当てを、表 A5-12 に示します。

この空間に含まれる他のエンコードは未定義です。

表 A5-12 同期化基本命令

op	命令	参照先	バリアント
0x00	ワードスワップ、バイトスワップ	*SWP*、*SWPB* P. A8-432 [a]	すべて
1000	排他レジスタストア	*STREX* P. A8-400	v6
1001	排他レジスタロード	*LDREX* P. A8-142	v6
1010	排他レジスタストア ダブルワード	*STREXD* P. A8-404	v6K
1011	排他レジスタロード ダブルワード	*LDREXD* P. A8-146	v6K
1100	排他レジスタストア バイト	*STREXB* P. A8-402	v6K
1101	排他レジスタロード バイト	*LDREXB* P. A8-144	v6K
1110	排他レジスタストア ハーフワード	*STREXH* P. A8-406	v6K
1111	排他レジスタロード ハーフワード	*LDREXH* P. A8-148	v6K

a. これらの命令の使用は推奨されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.11 MSR (immediate), and hints

31 30 29 28	27 26 25 24 23	22	21 20	19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
cond	0 0 1 1 0	op	1 0	op1		op2

Table A5-13 shows the allocation of encodings in this space.

Other encodings in this space are unallocated hints. They execute as NOPs, but software must not use them.

Table A5-13 MSR (immediate), and hints

op	op1	op2	Instruction	See	Variant
0	0000	00000000	No Operation hint	*NOP* on page A8-222	v6K, v6T2
		00000001	Yield hint	*YIELD* on page A8-812	v6K
		00000010	Wait For Event hint	*WFE* on page A8-808	v6K
		00000011	Wait For Interrupt hint	*WFI* on page A8-810	v6K
		00000100	Send Event hint	*SEV* on page A8-316	v6K
		1111xxxx	Debug hint	*DBG* on page A8-88	v7
	0100	-	Move to Special Register, application level	*MSR (immediate)* on page A8-208	All
	1x00	-			
	xx01	-	Move to Special Register, system level	*MSR (immediate)* on page B6-12	All
	xx1x	-			
1	-	-	Move to Special Register, system level	*MSR (immediate)* on page B6-12	All

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.11 MSR（イミディエート）およびヒント

31 30 29 28	27 26 25 24 23	22	21 20	19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
cond	0 0 1 1 0	op	1 0	op1		op2

この空間でのエンコードの割り当てを、表 A5-13 に示します。

この空間に含まれる他のエンコードは未割り当てのヒントです。NOP として実行されますが、ソフトウェアでは使用しないで下さい。

表 A5-13 MSR（イミディエート）およびヒント

op	op1	op2	命令	参照先	バリアント
0	0000	00000000	無操作ヒント	*NOP* P. A8-222	v6K、v6T2
		00000001	イールドヒント	*YIELD* P. A8-812	v6K
		00000010	イベント待ちヒント	*WFE* P. A8-808	v6K
		00000011	割り込み待ちヒント	*WFI* P. A8-810	v6K
		00000100	イベント送信ヒント	*SEV* P. A8-316	v6K
		1111xxxx	デバッグヒント	*DBG* P. A8-88	v7
	0100	–	特殊レジスタへの移動、アプリケーションレベル	*MSR（イミディエート）* P. A8-208	すべて
	1x00	–			
	xx01	–	特殊レジスタへの移動、システムレベル	*MSR（イミディエート）* P. B6-12	すべて
	xx1x	–			
1	–	–	特殊レジスタへの移動、システムレベル	*MSR（イミディエート）* P. B6-12	すべて

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.12 Miscellaneous instructions

31 30 29 28	27 26 25	24 23	22 21	20	19 18 17 16	15 14 13 12 11 10 9 8	7	6 5 4	3 2 1 0
cond	0 0 0	1 0	op	0	op1		0	op2	

Table A5-14 shows the allocation of encodings in this space.

Other encodings in this space are UNDEFINED.

Table A5-14 Miscellaneous instructions

op2	op	op1	Instruction or instruction class	See	Variant
000	x0	xxxx	Move Special Register to Register	*MRS* on page A8-206 *MRS* on page B6-10	All
	01	xx00	Move to Special Register, application level	*MSR (register)* on page A8-210	All
		xx01 xx1x	Move to Special Register, system level	*MSR (register)* on page B6-14	All
	11	-	Move to Special Register, system level	*MSR (register)* on page B6-14	All
001	01	-	Branch and Exchange	*BX* on page A8-62	v4T
	11	-	Count Leading Zeros	*CLZ* on page A8-72	v6
010	01	-	Branch and Exchange Jazelle	*BXJ* on page A8-64	v5TEJ
011	01	-	Branch with Link and Exchange	*BLX (register)* on page A8-60	v5T
101	-	-	Saturating addition and subtraction	*Saturating addition and subtraction* on page A5-13	-
111	01	-	Breakpoint	*BKPT* on page A8-56	v5T
	11	-	Secure Monitor Call	*SMC (previously SMI)* on page B6-18	Security Extensions

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.2.12 その他の命令

31 30 29 28	27 26 25	24 23	22 21	20	19 18 17 16	15 14 13 12 11 10 9 8	7	6 5 4	3 2 1 0
cond	0 0 0	1 0	op	0	op1		0	op2	

この空間でのエンコードの割り当てを、表 A5-14 に示します。

この空間に含まれる他のエンコードは未定義です。

表 A5-14 その他の命令

op2	op	op1	命令または命令クラス	参照先	バリアント
000	x0	xxxx	特殊レジスタのレジスタへの移動	*MRS* P. A8-206 *MRS* P. B6-10	すべて
	01	xx00	特殊レジスタへの移動、アプリケーションレベル	*MSR（レジスタ）* P. A8-210	すべて
		xx01 xx1x	特殊レジスタへの移動、システムレベル	*MSR（レジスタ）* P. B6-14	すべて
	11	–	特殊レジスタへの移動、システムレベル	*MSR（レジスタ）* P. B6-14	すべて
001	01	–	分岐交換	*BX* P. A8-62	v4T
	11	–	先行ゼロカウント	*CLZ* P. A8-72	v6
010	01	–	分岐と Jazelle 状態遷移	*BXJ* P. A8-64	v5TEJ
011	01	–	リンク付き分岐交換	*BLX（レジスタ）* P. A8-60	v5T
101	–	–	飽和加算および減算	*飽和加算および減算* P. A5-13	–
111	01	–	ブレークポイント	*BKPT* P. A8-56	v5T
	11	–	セキュア モニタコール	*SMC（以前の SMI）* P. B6-18	セキュリティ拡張機能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.3 Load/store word and unsigned byte

31 30 29 28	27 26	25	24 23 22 21	20 19 18 17	16 15 14 13 12 11 10 9 8 7 6 5	4	3 2 1 0
cond	0 1	A	op1	Rn		B	

These instructions have either A == 0 or B == 0. For instructions with A == 1 and B == 1, see *Media instructions* on page A5-21.

Table A5-15 shows the allocation of encodings in this space. These encodings are in all architecture variants.

Table A5-15 Single data transfer instructions

A	op1	B	Rn	Instruction	See
0	xx0x0 not 0x010	-	-	Store Register	*STR (immediate, ARM)* on page A8-384
1	xx0x0 not 0x010	0	-	Store Register	*STR (register)* on page A8-386
0	0x010	-	-	Store Register Unprivileged	*STRT* on page A8-416
1	0x010	0	-		
0	xx0x1 not 0x011	-	not 1111	Load Register (immediate)	*LDR (immediate, ARM)* on page A8-120
	xx0x1 not 0x011	-	1111	Load Register (literal)	*LDR (literal)* on page A8-122
1	xx0x1 not 0x011	0	-	Load Register	*LDR (register)* on page A8-124
0	0x011	-	-	Load Register Unprivileged	*LDRT* on page A8-176
1	0x011	0	-		
0	xx1x0 not 0x110	-	-	Store Register Byte (immediate)	*STRB (immediate, ARM)* on page A8-390
1	xx1x0 not 0x110	0	-	Store Register Byte (register)	*STRB (register)* on page A8-392
0	0x110	-	-	Store Register Byte Unprivileged	*STRBT* on page A8-394
1	0x110	0	-		
0	xx1x1 not 0x111	-	not 1111	Load Register Byte (immediate)	*LDRB (immediate, ARM)* on page A8-128
	xx1x1 not 0x111	-	1111	Load Register Byte (literal)	*LDRB (literal)* on page A8-130

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A5.3 ワードおよび符号なしバイトのロード / ストア

31 30 29 28	27 26	25	24 23 22 21	20 19 18 17	16 15 14 13 12 11 10 9 8 7 6 5	4	3 2 1 0
cond	0 1	A	op1	Rn		B	

これらの命令には A == 0 または B == 0 のいずれかが含まれます。A == 1 および B == 1 の命令については、*メディア命令* P. A5-21 を参照して下さい。

この空間でのエンコードの割り当てを、表 A5-15 に示します。これらのエンコードは、すべてのアーキテクチャバリアントに存在します。

表 A5-15 単一データ転送命令

A	op1	B	Rn	命令	参照先
0	0x010 以外の xx0x0	–	–	レジスタストア	*STR（イミディエート、ARM）* P. A8-384
1	0x010 以外の xx0x0	0	–	レジスタストア	*STR（レジスタ）* P. A8-386
0	0x010	–	–	非特権レジスタストア	*STRT* P. A8-416
1	0x010	0	–		
0	0x011 以外の xx0x1	–	1111 以外	レジスタロード（イミディエート）	*LDR（イミディエート、ARM）* P. A8-120
	0x011 以外の xx0x1	–	1111	レジスタロード（リテラル）	*LDR（リテラル）* P. A8-122
1	0x011 以外の xx0x1	0	–	レジスタロード	*LDR（レジスタ）* P. A8-124
0	0x011	–	–	非特権レジスタロード	*LDRT* P. A8-176
1	0x011	0	–		
0	0x110 以外の xx1x0	–	–	レジスタストア バイト（イミディエート）	*STRB（イミディエート、ARM）* P. A8-390
1	0x110 以外の xx1x0	0	–	レジスタストア バイト（レジスタ）	*STRB（レジスタ）* P. A8-392
0	0x110	–	–	非特権レジスタストア バイト	*STRBT* P. A8-394
1	0x110	0	–		

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A5-15 Single data transfer instructions (continued)

A	op1	B	Rn	Instruction	See
1	xx1x1 not 0x111	0	-	Load Register Byte (register)	*LDRB (register)* on page A8-132
0	0x111	-	-	Load Register Byte Unprivileged	*LDRBT* on page A8-134
1	0x111	0	-		

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A5-15 単一データ転送命令（続き）

A	op1	B	Rn	命令	参照先
0	0x111 以外の xx1x1	–	1111 以外	レジスタロード バイト（イミディエート）	*LDRB（イミディエート、ARM）* P. A8-128
	0x111 以外の xx1x1	–	1111	レジスタロード バイト（リテラル）	*LDRB（リテラル）* P. A8-130
1	0x111 以外の xx1x1	0	–	レジスタロード バイト（レジスタ）	*LDRB（レジスタ）* P. A8-132
0	0x111	–	–	非特権レジスタロード バイト	*LDRBT* P. A8-134
1	0x111	0	–		

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.4 Media instructions

31 30 29 28	27 26 25	24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
cond	0 1 1	op1		Rd		op2	1	Rn

Table A5-16 shows the allocation of encodings in this space.

Other encodings in this space are UNDEFINED.

Table A5-16 Media instructions

op1	op2	Rd	Rn	Instructions	See	Variant
000xx	-	-	-	-	*Parallel addition and subtraction, signed* on page A5-22	-
001xx	-	-	-	-	*Parallel addition and subtraction, unsigned* on page A5-23	-
01xxx	-	-	-	-	*Packing, unpacking, saturation, and reversal* on page A5-24	-
10xxx	-	-	-	-	*Signed multiplies* on page A5-26	-
11000	000	1111	-	Unsigned Sum of Absolute Differences	*USAD8* on page A8-500	v6
	000	not 1111	-	Unsigned Sum of Absolute Differences and Accumulate	*USADA8* on page A8-502	v6
1101x	x10	-	-	Signed Bit Field Extract	*SBFX* on page A8-308	v6T2
1110x	x00	-	1111	Bit Field Clear	*BFC* on page A8-46	v6T2
		-	not 1111	Bit Field Insert	*BFI* on page A8-48	v6T2
1111x	x10	-	-	Unsigned Bit Field Extract	*UBFX* on page A8-466	v6T2
11111	111	-	-	Permanently UNDEFINED. This space will not be allocated in future.		

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A5.4 メディア命令

31 30 29 28	27 26 25	24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
cond	0 1 1	op1		Rd		op2	1	Rn

この空間でのエンコードの割り当てを、表 A5-16 に示します。

この空間に含まれる他のエンコードは未定義です。

表 A5-16 メディア命令

op1	op2	Rd	Rn	命令	参照先	バリアント
000xx	–	–	–	–	*並列加算および減算、符号付き* P. A5-22	–
001xx	–	–	–	–	*並列加算および減算、符号なし* P. A5-23	–
01xxx	–	–	–	–	*パッキング、アンパッキング、飽和、反転* P. A5-24	–
10xxx	–	–	–	–	*符号付き乗算* P. A5-26	–
11000	000	1111	–	符号なし絶対差の和	*USAD8* P. A8-500	v6
	000	1111 以外	–	符号なし絶対差の和および累積	*USADA8* P. A8-502	v6
1101x	x10	–	–	符号付きビットフィールドの抽出	*SBFX* P. A8-308	v6T2
1110x	x00	–	1111	ビッドフィールドのクリア	*BFC* P. A8-46	v6T2
		–	1111 以外	ビットフィールドの挿入	*BFI* P. A8-48	v6T2
1111x	x10	–	–	符号なしビットフィールドの抽出	*UBFX* P. A8-466	v6T2
11111	111	–	–	恒久的に未定義。この空間は将来も割り当てられません。		

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.4.1 Parallel addition and subtraction, signed

31 30 29 28	27 26 25 24 23 22	21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5	4	3 2 1 0
cond	0 1 1 0 0 0	op1		op2	1	

Table A5-17 shows the allocation of encodings in this space. These encodings are all available in ARMv6 and above, and are UNDEFINED in earlier variants of the architecture.

Other encodings in this space are UNDEFINED.

Table A5-17 Signed parallel addition and subtraction instructions

op1	op2	Instruction	See
01	000	Add 16-bit	*SADD16* on page A8-296
01	001	Add and Subtract with Exchange	*SASX* on page A8-300
01	010	Subtract and Add with Exchange	*SSAX* on page A8-366
01	011	Subtract 16-bit	*SSUB16* on page A8-368
01	100	Add 8-bit	*SADD8* on page A8-298
01	111	Subtract 8-bit	*SSUB8* on page A8-370
Saturating instructions			
10	000	Saturating Add 16-bit	*QADD16* on page A8-252
10	001	Saturating Add and Subtract with Exchange	*QASX* on page A8-256
10	010	Saturating Subtract and Add with Exchange	*QSAX* on page A8-262
10	011	Saturating Subtract 16-bit	*QSUB16* on page A8-266
10	100	Saturating Add 8-bit	*QADD8* on page A8-254
10	111	Saturating Subtract 8-bit	*QSUB8* on page A8-268
Halving instructions			
11	000	Halving Add 16-bit	*SHADD16* on page A8-318
11	001	Halving Add and Subtract with Exchange	*SHASX* on page A8-322
11	010	Halving Subtract and Add with Exchange	*SHSAX* on page A8-324
11	011	Halving Subtract 16-bit	*SHSUB16* on page A8-326
11	100	Halving Add 8-bit	*SHADD8* on page A8-320
11	111	Halving Subtract 8-bit	*SHSUB8* on page A8-328

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.4.1 並列加算および減算、符号付き

31 30 29 28	27 26 25 24 23 22	21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5	4	3 2 1 0
cond	0 1 1 0 0 0	op1		op2	1	

この空間でのエンコードの割り当てを、表 A5-17 に示します。これらのエンコードはすべて、ARMv6 およびそれ以降で使用できます。それ以前のアーキテクチャバリアントでは未定義です。

この空間に含まれる他のエンコードは未定義です。

表 A5-17 符号付き並列加算および減算命令

op1	op2	命令	参照先
01	000	16 ビット加算	*SADD16* P. A8-296
01	001	交換付き加算および減算	*SASX* P. A8-300
01	010	交換付き減算および加算	*SSAX* P. A8-366
01	011	16 ビット減算	*SSUB16* P. A8-368
01	100	8 ビット加算	*SADD8* P. A8-298
01	111	8 ビット減算	*SSUB8* P. A8-370
飽和命令			
10	000	飽和 16 ビット加算	*QADD16* P. A8-252
10	001	交換付き飽和加算および減算	*QASX* P. A8-256
10	010	交換付き飽和減算および加算	*QSAX* P. A8-262
10	011	飽和 16 ビット減算	*QSUB16* P. A8-266
10	100	飽和 8 ビット加算	*QADD8* P. A8-254
10	111	飽和 8 ビット減算	*QSUB8* P. A8-268
半減命令			
11	000	16 ビット加算半減	*SHADD16* P. A8-318
11	001	交換付き加算および減算半減	*SHASX* P. A8-322
11	010	交換付き減算および加算半減	*SHSAX* P. A8-324
11	011	16 ビット減算半減	*SHSUB16* P. A8-326
11	100	8 ビット加算半減	*SHADD8* P. A8-320
11	111	8 ビット減算半減	*SHSUB8* P. A8-328

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.4.2 Parallel addition and subtraction, unsigned

31 30 29 28	27 26 25 24 23 22	21 20 19	18 17 16 15 14 13 12 11 10 9 8	7 6 5	4	3 2 1 0
cond	0 1 1 0 0 1	op1		op2	1	

Table A5-18 shows the allocation of encodings in this space. These encodings are all available in ARMv6 and above, and are UNDEFINED in earlier variants of the architecture.

Other encodings in this space are UNDEFINED.

Table A5-18 Unsigned parallel addition and subtractions instructions

op1	op2	Instruction	See
01	000	Add 16-bit	*UADD16* on page A8-460
01	001	Add and Subtract with Exchange	*UASX* on page A8-464
01	010	Subtract and Add with Exchange	*USAX* on page A8-508
01	011	Subtract 16-bit	*USUB16* on page A8-510
01	100	Add 8-bit	*UADD8* on page A8-462
01	111	Subtract 8-bit	*USUB8* on page A8-512
Saturating instructions			
10	000	Saturating Add 16-bit	*UQADD16* on page A8-488
10	001	Saturating Add and Subtract with Exchange	*UQASX* on page A8-492
10	010	Saturating Subtract and Add with Exchange	*UQSAX* on page A8-494
10	011	Saturating Subtract 16-bit	*UQSUB16* on page A8-496
10	100	Saturating Add 8-bit	*UQADD8* on page A8-490
10	111	Saturating Subtract 8-bit	*UQSUB8* on page A8-498
Halving instructions			
11	000	Halving Add 16-bit	*UHADD16* on page A8-470
11	001	Halving Add and Subtract with Exchange	*UHASX* on page A8-474
11	010	Halving Subtract and Add with Exchange	*UHSAX* on page A8-476
11	011	Halving Subtract 16-bit	*UHSUB16* on page A8-478
11	100	Halving Add 8-bit	*UHADD8* on page A8-472
11	111	Halving Subtract 8-bit	*UHSUB8* on page A8-480

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.4.2 並列加算および減算、符号なし

31 30 29 28	27 26 25 24 23 22	21 20 19	18 17 16 15 14 13 12 11 10 9 8	7 6 5	4	3 2 1 0
cond	0 1 1 0 0 1	op1		op2	1	

この空間でのエンコードの割り当てを、表 A5-18 に示します。これらのエンコードはすべて、ARMv6 およびそれ以降で使用できます。それ以前のアーキテクチャバリアントでは未定義です。

この空間に含まれる他のエンコードは未定義です。

表 A5-18 符号なし並列加算および減算命令

op1	op2	命令	参照先
01	000	16 ビット加算	*UADD16* P. A8-460
01	001	交換付き加算および減算	*UASX* P. A8-464
01	010	交換付き減算および加算	*USAX* P. A8-508
01	011	16 ビット減算	*USUB16* P. A8-510
01	100	8 ビット加算	*UADD8* P. A8-462
01	111	8 ビット減算	*USUB8* P. A8-512
飽和命令			
10	000	飽和 16 ビット加算	*UQADD16* P. A8-488
10	001	交換付き飽和加算および減算	*UQASX* P. A8-492
10	010	交換付き飽和減算および加算	*UQSAX* P. A8-494
10	011	飽和 16 ビット減算	*UQSUB16* P. A8-496
10	100	飽和 8 ビット加算	*UQADD8* P. A8-490
10	111	飽和 8 ビット減算	*UQSUB8* P. A8-498
半減命令			
11	000	16 ビット加算半減	*UHADD16* P. A8-470
11	001	交換付き加算および減算半減	*UHASX* P. A8-474
11	010	交換付き減算および加算半減	*UHSAX* P. A8-476
11	011	16 ビット減算半減	*UHSUB16* P. A8-478
11	100	8 ビット加算半減	*UHADD8* P. A8-472
11	111	8 ビット減算半減	*UHSUB8* P. A8-480

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.4.3 Packing, unpacking, saturation, and reversal

31 30 29 28	27 26 25 24 23	22 21 20	19 18 17 16	15 14 13 12 11 10 9 8	7 6 5	4	3 2 1 0
cond	0 1 1 0 1	op1	A		op2	1	

Table A5-19 shows the allocation of encodings in this space.

Other encodings in this space are UNDEFINED.

Table A5-19 Packing, unpacking, saturation, and reversal instructions

op1	op2	A	Instructions	See	Variant
000	xx0	-	Pack Halfword	*PKH* on page A8-234	v6
01x	xx0	-	Signed Saturate	*SSAT* on page A8-362	v6
11x	xx0	-	Unsigned Saturate	*USAT* on page A8-504	v6
000	011	not 1111	Signed Extend and Add Byte 16	*SXTAB16* on page A8-436	v6
		1111	Signed Extend Byte 16	*SXTB16* on page A8-442	v6
	101	-	Select Bytes	*SEL* on page A8-312	v6
010	001	-	Signed Saturate 16	*SSAT16* on page A8-364	v6
	011	not 1111	Signed Extend and Add Byte	*SXTAB* on page A8-434	v6
		1111	Signed Extend Byte	*SXTB* on page A8-440	v6
011	001	-	Byte-Reverse Word	*REV* on page A8-272	v6
	011	not 1111	Signed Extend and Add Halfword	*SXTAH* on page A8-438	v6
		1111	Signed Extend Halfword	*SXTH* on page A8-444	v6
011	101	-	Byte-Reverse Packed Halfword	*REV16* on page A8-274	v6
100	011	not 1111	Unsigned Extend and Add Byte 16	*UXTAB16* on page A8-516	v6
		1111	Unsigned Extend Byte 16	*UXTB16* on page A8-522	v6
110	001	-	Unsigned Saturate 16	*USAT16* on page A8-506	v6
	011	not 1111	Unsigned Extend and Add Byte	*UXTAB* on page A8-514	v6
		1111	Unsigned Extend Byte	*UXTB* on page A8-520	v6

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.4.3 パッキング、アンパッキング、飽和、反転

31 30 29 28	27 26 25 24 23	22 21 20	19 18 17 16	15 14 13 12 11 10 9 8	7 6 5	4	3 2 1 0
cond	0 1 1 0 1	op1	A		op2	1	

この空間でのエンコードの割り当てを、表 A5-19 に示します。

この空間に含まれる他のエンコードは未定義です。

表 A5-19 パッキング、アンパッキング、飽和、反転命令

op1	op2	A	命令	参照先	バリアント
000	xx0	–	パックハーフワード	*PKH* P. A8-234	v6
01x	xx0	–	符号付き飽和	*SSAT* P. A8-362	v6
11x	xx0	–	符号なし飽和	*USAT* P. A8-504	v6
000	011	1111 以外	符号付きバイト拡張および加算 16	*SXTAB16* P. A8-436	v6
		1111	符号付きバイト拡張 16	*SXTB16* P. A8-442	v6
	101	–	バイト選択	*SEL* P. A8-312	v6
010	001	–	符号付き飽和 16	*SSAT16* P. A8-364	v6
	011	1111 以外	符号付きバイト拡張および加算	*SXTAB* P. A8-434	v6
		1111	符号付きバイト拡張	*SXTB* P. A8-440	v6
011	001	–	ワードのバイト反転	*REV* P. A8-272	v6
	011	1111 以外	符号付きハーフワード拡張および加算	*SXTAH* P. A8-438	v6
		1111	符号付きハーフワード拡張	*SXTH* P. A8-444	v6
011	101	–	パッキングされたハーフワードのバイト反転	*REV16* P. A8-274	v6
100	011	1111 以外	符号なしバイト拡張および加算 16	*UXTAB16* P. A8-516	v6
		1111	符号なしバイト拡張 16	*UXTB16* P. A8-522	v6
110	001	–	符号なし飽和 16	*USAT16* P. A8-506	v6
	011	1111 以外	符号なしバイト拡張および加算	*UXTAB* P. A8-514	v6
		1111	符号なしバイト拡張	*UXTB* P. A8-520	v6

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A5-19 Packing, unpacking, saturation, and reversal instructions (continued)

op1	op2	A	Instructions	See	Variant
111	001	-	Reverse Bits	*RBIT* on page A8-270	v6T2
	011	not 1111	Unsigned Extend and Add Halfword	*UXTAH* on page A8-518	v6
		1111	Unsigned Extend Halfword	*UXTH* on page A8-524	v6
	101	-	Byte-Reverse Signed Halfword	*REVSH* on page A8-276	v6

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A5-19 パッキング、アンパッキング、飽和、反転命令（続き）

op1	op2	A	命令	参照先	バリアント
111	001	–	ビット反転	*RBIT* P. A8-270	v6T2
	011	1111 以外	符号なしハーフワード拡張および加算	*UXTAH* P. A8-518	v6
		1111	符号なしハーフワード拡張	*UXTH* P. A8-524	v6
	101	–	符号付きハーフワードのバイト反転	*REVSH* P. A8-276	v6

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.4.4 Signed multiplies

31 30 29 28	27 26 25 24 23	22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
cond	0 1 1 1 0	op1		A		op2	1	

Table A5-20 shows the allocation of encodings in this space. These encodings are all available in ARMv6T2 and above, and are UNDEFINED in earlier variants of the architecture.

Other encodings in this space are UNDEFINED.

Table A5-20 Signed multiply instructions

op1	op2	A	Instruction	See
000	00x	not 1111	Signed Multiply Accumulate Dual	*SMLAD* on page A8-332
		1111	Signed Dual Multiply Add	*SMUAD* on page A8-352
	01x	not 1111	Signed Multiply Subtract Dual	*SMLSD* on page A8-342
		1111	Signed Dual Multiply Subtract	*SMUSD* on page A8-360
100	00x	-	Signed Multiply Accumulate Long Dual	*SMLALD* on page A8-338
	01x	-	Signed Multiply Subtract Long Dual	*SMLSLD* on page A8-344
101	00x	not 1111	Signed Most Significant Word Multiply Accumulate	*SMMLA* on page A8-346
		1111	Signed Most Significant Word Multiply	*SMMUL* on page A8-350
	11x	-	Signed Most Significant Word Multiply Subtract	*SMMLS* on page A8-348

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.4.4 符号付き乗算

31 30 29 28	27 26 25 24 23	22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
cond	0 1 1 1 0	op1		A		op2	1	

この空間でのエンコードの割り当てを、表 A5-20 に示します。これらのエンコードは、すべて ARMv6T2 およびそれ以降で使用できます。それ以前のアーキテクチャバリアントでは未定義です。

この空間に含まれる他のエンコードは未定義です。

表 A5-20 符号付き乗算命令

op1	op2	A	命令	参照先
000	00x	1111 以外	符号付きデュアル積和演算	*SMLAD* P. A8-332
		1111	符号付きデュアル積加算	*SMUAD* P. A8-352
	01x	1111 以外	符号付きデュアル積減算	*SMLSD* P. A8-342
		1111	符号付きデュアル積減算	*SMUSD* P. A8-360
100	00x	–	符号付きデュアルロング積和演算	*SMLALD* P. A8-338
	01x	–	符号付きデュアルロング積減算	*SMLSLD* P. A8-344
101	00x	1111 以外	符号付き最上位ワード積和演算	*SMMLA* P. A8-346
		1111	符号付き最上位ワード乗算	*SMMUL* P. A8-350
	11x	–	符号付き最上位ワード積減算	*SMMLS* P. A8-348

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.5 Branch, branch with link, and block data transfer

31 30 29 28	27 26	25 24 23 22 21 20 19 18 17 16	15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0	op	R	

Table A5-21 shows the allocation of encodings in this space. These encodings are in all architecture variants.

Table A5-21 Branch, branch with link, and block data transfer instructions

op	R	Instructions	See
0000x0	-	Store Multiple Decrement After	*STMDA / STMED* on page A8-376
0000x1	-	Load Multiple Decrement After	*LDMDA / LDMFA* on page A8-112
0010x0	-	Store Multiple (Increment After)	*STM / STMIA / STMEA* on page A8-374
0010x1	-	Load Multiple (Increment After)	*LDM / LDMIA / LDMFD* on page A8-110
0100x0	-	Store Multiple Decrement Before	*STMDB / STMFD* on page A8-378
0100x1	-	Load Multiple Decrement Before	*LDMDB / LDMEA* on page A8-114
0110x0	-	Store Multiple Increment Before	*STMIB / STMFA* on page A8-380
0110x1	-	Load Multiple Increment Before	*LDMIB / LDMED* on page A8-116
0xx1x0	-	Store Multiple (user registers)	*STM (user registers)* on page B6-22
0xx1x1	0	Load Multiple (user registers)	*LDM (user registers)* on page B6-7
	1	Load Multiple (exception return)	*LDM (exception return)* on page B6-5
10xxxx	-	Branch	*B* on page A8-44
11xxxx	-	Branch with Link	*BL, BLX (immediate)* on page A8-58

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.5 分岐、リンク付き分岐、ブロックデータ転送

31 30 29 28	27 26	25 24 23 22 21 20 19 18 17 16	15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0	op	R	

この空間でのエンコードの割り当てを、表 A5-21 に示します。これらのエンコードは、すべてのアーキテクチャバリアントに存在します。

表 A5-21 分岐、リンク付き分岐、ブロックデータ転送命令

op	R	命令	参照先
0000x0	–	ポストデクリメント複数ストア	*STMDA*、*STMED* P. A8-376
0000x1	–	ポストデクリメント複数ロード	*LDMDA/LDMFA* P. A8-112
0010x0	–	複数ストア（ポストインクリメント）	*STM*、*STMIA*、*STMEA* P. A8-374
0010x1	–	複数ロード（ポストインクリメント）	*LDM/LDMIA/LDMFD* P. A8-110
0100x0	–	プリデクリメント複数ストア	*STMDB*、*STMFD* P. A8-378
0100x1	–	プリデクリメント複数ロード	*LDMDB/LDMEA* P. A8-114
0110x0	–	プリインクリメント複数ストア	*STMIB*、*STMFA* P. A8-380
0110x1	–	プリインクリメント複数ロード	*LDMIB/LDMED* P. A8-116
0xx1x0	–	複数ストア（ユーザレジスタ）	*STM（ユーザレジスタ）* P. B6-22
0xx1x1	0	複数ロード（ユーザレジスタ）	*LDM（ユーザレジスタ）* P. B6-7
	1	複数ロード（例外からの復帰）	*LDM（例外からの復帰）* P. B6-5
10xxxx	–	分岐	*B* P. A8-44
11xxxx	–	リンク付き分岐	*BL*、*BLX（イミディエート）* P. A8-58

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.6 Supervisor Call, and coprocessor instructions

31 30 29 28	27 26	25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
cond	1 1	op1	Rn		coproc		op	

Table A5-22 shows the allocation of encodings in this space.

Table A5-22 Supervisor Call, and coprocessor instructions

op1	op	coproc	Rn	Instructions	See	Variant
0xxxxx[a]	-	101x	-	Advanced SIMD, VFP	*Extension register load/store instructions* on page A7-26	
0xxxx0[a]	-	not 101x	-	Store Coprocessor	*STC, STC2* on page A8-372	All
0xxxx1[a]	-	not 101x	not 1111	Load Coprocessor	*LDC, LDC2 (immediate)* on page A8-106	All
			1111	Load Coprocessor	*LDC, LDC2 (literal)* on page A8-108	All
00000x	-	-	-	UNDEFINED	-	-
00010x	-	101x	-	Advanced SIMD, VFP	*64-bit transfers between ARM core and extension registers* on page A7-32	
000100	-	not 101x	-	Move to Coprocessor from two ARM core registers	*MCRR, MCRR2* on page A8-188	v5TE
000101	-	not 101x	-	Move to two ARM core registers from Coprocessor	*MRRC, MRRC2* on page A8-204	v5TE
10xxxx	0	101x	-	-	*VFP data-processing instructions* on page A7-24	
		not 101x	-	Coprocessor data operations	*CDP, CDP2* on page A8-68	All
	1	101x	-	Advanced SIMD, VFP	*8, 16, and 32-bit transfer between ARM core and extension registers* on page A7-31	
10xxx0	1	not 101x	-	Move to Coprocessor from ARM core register	*MCR, MCR2* on page A8-186	All

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.6 スーパバイザコールおよびコプロセッサ命令

31 30 29 28	27 26	25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
cond	1 1	op1	Rn		coproc		op	

この空間でのエンコードの割り当てを、表 A5-22 に示します。

表 A5-22 スーパバイザコールおよびコプロセッサ命令

op1	op	coproc	Rn	命令	参照先	バリアント
0xxxxx[a]	–	101x	–	アドバンスト SIMD、VFP	*拡張機能レジスタロード / ストア命令* P. A7-26	
0xxxx0[a]	–	101x 以外	–	コプロセッサストア	*STC*、*STC2* P. A8-372	すべて
0xxxx1[a]	–	101x 以外	1111 以外	コプロセッサロード	*LDC*、*LDC2（イミディエート）* P. A8-106	すべて
			1111	コプロセッサロード	*LDC*、*LDC2（リテラル）* P. A8-108	すべて
00000x	–	–	–	未定義	–S	–
00010x	–	101x	–	アドバンスト SIMD、VFP	*ARM コアレジスタと拡張機能レジスタとの 64 ビット転送* P. A7-32	
000100	–	101x 以外	–	2 つの ARM コアレジスタからコプロセッサへの移動	*MCRR*、*MCRR2* P. A8-188	v5TE
000101	–	not 101x	–	コプロセッサから 2 つの ARM コアレジスタへの移動	*MRRC*、*MRRC2* P. A8-204	v5TE
10xxxx	0	101x	–	–	*VFP のデータ処理命令* P. A7-24	
		101x 以外	–	コプロセッサデータ演算	*CDP*、*CDP2* P. A8-68	すべて
	1	101x	–	アドバンスト SIMD、VFP	*ARM コアレジスタと拡張機能レジスタとの 8、16、32 ビット転送* P. A7-31	
10xxx0	1	101x 以外	–	ARM コアレジスタからコプロセッサへの移動	*MCR*、*MCR2* P. A8-186	すべて

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A5-22 Supervisor Call, and coprocessor instructions (continued)

op1	op	coproc	Rn	Instructions	See	Variant
10xxx1	1	not 101x	-	Move to ARM core register from Coprocessor	*MRC, MRC2* on page A8-202	All
11xxxx	-	-	-	Supervisor Call	*SVC (previously SWI)* on page A8-430	All

a. But not 000x0x

For more information about specific coprocessors see *Coprocessor support* on page A2-68.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A5-22 スーパバイザコールおよびコプロセッサ命令（続き）

op1	op	coproc	Rn	命令	参照先	バリアント
10xxx1	1	101x以外	–	コプロセッサから ARM コアレジスタへの移動	*MRC*、*MRC2* P. A8-202	すべて
11xxxx	–	–	–	スーパバイザコール	*SVC（以前の SWI）* P. A8-430	すべて

a. ただし 000x0x 以外

各コプロセッサの詳細については、*コプロセッサのサポート* P. A2-68 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.7 Unconditional instructions

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5	4	3 2 1 0
1 1 1 1	op1	Rn		op	

Table A5-23 shows the allocation of encodings in this space.

Other encodings in this space are UNDEFINED in ARMv5 and above.

All encodings in this space are UNPREDICTABLE in ARMv4 and ARMv4T.

Table A5-23 Unconditional instructions

op1	op	Rn	Instruction	See	Variant
0xxxxxxx	-	-	-	*Miscellaneous instructions, memory hints, and Advanced SIMD instructions* on page A5-31	
100xx1x0	-	-	Store Return State	*SRS* on page B6-20	v6
100xx0x1	-	-	Return From Exception	*RFE* on page B6-16	v6
101xxxxx	-	-	Branch with Link and Exchange	*BL, BLX (immediate)* on page A8-58	v5
11000x11	-	not 1111	Load Coprocessor (immediate)	*LDC, LDC2 (immediate)* on page A8-106	v5
11001xx1	-	1111	Load Coprocessor (literal)	*LDC, LDC2 (literal)* on page A8-108	v5
1101xxx1	-	1111			
11000x10 11001xx0 1101xxx0	-	-	Store Coprocessor	*STC, STC2* on page A8-372	v5
11000100	-	-	Move to Coprocessor from two ARM core registers	*MCRR, MCRR2* on page A8-188	v6
11000101	-	-	Move to two ARM core registers from Coprocessor	*MRRC, MRRC2* on page A8-204	v6
1110xxxx	0	-	Coprocessor data operations	*CDP, CDP2* on page A8-68	v5
1110xxx0	1	-	Move to Coprocessor from ARM core register	*MCR, MCR2* on page A8-186	v5
1110xxx1	1	-	Move to ARM core register from Coprocessor	*MRC, MRC2* on page A8-202	v5

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.7 無条件命令

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5	4	3 2 1 0
1 1 1 1	op1	Rn		op	

この空間でのエンコードの割り当てを、表 A5-23 に示します。

この空間に含まれる他のエンコードは、ARMv5 およびそれ以降では未定義です。

この空間に含まれるすべてのエンコードは、ARMv4 および ARMv4T では予測不能です。

表 A5-23 無条件命令

op1	op	Rn	命令	参照先	バリアント
0xxxxxxx	–	–	–	*その他の命令*、*メモリヒント*、*アドバンスト SIMD 命令* P. A5-31	
100xx1x0	–	–	復帰状態ストア	*SRS* P. B6-20	v6
100xx0x1	–	–	例外からの復帰	*RFE* P. B6-16	v6
101xxxxx	–	–	リンク付き分岐交換	*BL*、*BLX（イミディエート）* P. A8-58	v5
11000x11	–	1111 以外	コプロセッサロード（イミディエート）	*LDC*、*LDC2（イミディエート）* P. A8-106	v5
11001xx1	–	1111	コプロセッサロード（リテラル）	*LDC*、*LDC2（リテラル）* P. A8-108	v5
1101xxx1	–	1111			
11000x10 11001xx0 1101xxx0	–	–	コプロセッサストア	*STC*、*STC2* P. A8-372	v5
11000100	–	–	2 つの ARM コアレジスタからコプロセッサへの移動	*MCRR*、*MCRR2* P. A8-188	v6
11000101	–	–	コプロセッサから 2 つの ARM コアレジスタへの移動	*MRRC*、*MRRC2* P. A8-204	v6
1110xxxx	0	–	コプロセッサデータ演算	*CDP*、*CDP2* P. A8-68	v5
1110xxx0	1	–	ARM コアレジスタからコプロセッサへの移動	*MCR*、*MCR2* P. A8-186	v5
1110xxx1	1	–	コプロセッサから ARM コアレジスタへの移動	*MRC*、*MRC2* P. A8-202	v5

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.7.1 Miscellaneous instructions, memory hints, and Advanced SIMD instructions

31 30 29 28 27	26 25 24 23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 0	op1	Rn		op2	

Table A5-24 shows the allocation of encodings in this space.

Other encodings in this space are UNDEFINED in ARMv5 and above. All these encodings are UNPREDICTABLE in ARMv4 and ARMv4T.

Table A5-24 Hints, and Advanced SIMD instructions

op1	op2	Rn	Instruction	See	Variant
0010000	xx0x	xxx0	Change Processor State	*CPS* on page B6-3	v6
0010000	0000	xxx1	Set Endianness	*SETEND* on page A8-314	v6
01xxxxx	-	-	See *Advanced SIMD data-processing instructions* on page A7-10		v7
100xxx0	-	-	See *Advanced SIMD element or structure load/store instructions* on page A7-27		v7
100x001	-	-	Unallocated memory hint (treat as NOP)		MP[a] Extensions
100x101	-	-	Preload Instruction	*PLI (immediate, literal)* on page A8-242	v7
101x001	-	not 1111	Preload Data with intent to Write	*PLD, PLDW (immediate)* on page A8-236	MP[a] Extensions
		1111	UNPREDICTABLE	-	-
101x101	-	not 1111	Preload Data	*PLD, PLDW (immediate)* on page A8-236	v5TE
		1111	Preload Data	*PLD (literal)* on page A8-238	v5TE
1010111	0001	-	Clear-Exclusive	*CLREX* on page A8-70	v6K
	0100	-	Data Synchronization Barrier	*DSB* on page A8-92	v6T2
	0101	-	Data Memory Barrier	*DMB* on page A8-90	v7
	0110	-	Instruction Synchronization Barrier	*ISB* on page A8-102	v6T2
10xxx11	-	-	UNPREDICTABLE except as shown above		-

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A5.7.1 その他の命令、メモリヒント、アドバンスト SIMD 命令

31 30 29 28 27	26 25 24 23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 0	op1	Rn		op2	

この空間でのエンコードの割り当てを、表 A5-24 に示します。

この空間に含まれる他のエンコードは、ARMv5 およびそれ以降では未定義です。これらすべてのエンコードは、ARMv4 と ARMv4T では予測不能です。

表 A5-24 ヒントおよびアドバンスト SIMD 命令

op1	op2	Rn	命令	参照先	バリアント
0010000	xx0x	xxx0	プロセッサ状態変更	*CPS* P. B6-3	v6
0010000	0000	xxx1	エンディアン形式の設定	*SETEND* P. A8-314	v6
01xxxxx	–	–	*アドバンスト SIMD のデータ処理命令* P. A7-10 を参照		v7
100xxx0	–	–	*アドバンスト SIMD のエレメントまたは構造ロード / ストア命令* P. A7-27 を参照		v7
100x001	–	–	未割り当てのメモリヒント（NOP として扱う）		MP[a] 拡張機能
100x101	–	–	命令のプリロード	*PLI（イミディエート、リテラル）* P. A8-242	v7
101x001	–	1111 以外	書き込みを意図したデータのプリロード	*PLD、PLDW（イミディエート）* P. A8-236	MP[a] 拡張機能
		1111	予測不能	–	–
101x101	–	1111 以外	データのプリロード	*PLD、PLDW（イミディエート）* P. A8-236	v5TE
		1111	データのプリロード	*PLD（リテラル）* P. A8-238	v5TE
1010111	0001	–	排他クリア	*CLREX* P. A8-70	v6K
	0100	–	データ同期バリア	*DSB* P. A8-92	v6T2
	0101	–	データ メモリバリア	*DMB* P. A8-90	v7
	0110	–	命令同期バリア	*ISB* P. A8-102	v6T2
10xxx11	–	–	上記以外は予測不能		–

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A5-24 Hints, and Advanced SIMD instructions (continued)

op1	op2	Rn	Instruction	See	Variant
110x001	xxx0	-	Unallocated memory hint (treat as NOP)		MP[a] Extensions
110x101	xxx0	-	Preload Instruction	*PLI (register)* on page A8-244	v7
111x001	xxx0	-	Preload Data with intent to Write	*PLD, PLDW (register)* on page A8-240	MP[a] Extensions
111x101	xxx0	-	Preload Data	*PLD, PLDW (register)* on page A8-240	v5TE
11xxx11	xxx0	-	UNPREDICTABLE	-	-

a. Multiprocessing Extensions.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A5-24 ヒントおよびアドバンスト SIMD 命令（続き）

op1	op2	Rn	命令	参照先	バリアント
110x001	xxx0	–	未割り当てのメモリヒント	（NOP として扱う）	MP[a] 拡張機能
110x101	xxx0	–	命令のプリロード	*PLI（レジスタ）* P. A8-244	v7
111x001	xxx0	–	書き込みを意図したデータのプリロード	*PLD*、*PLDW（レジスタ）* P. A8-240	MP[a] 拡張機能
111x101	xxx0	–	データのプリロード	*PLD*、*PLDW（レジスタ）* P. A8-240	v5TE
11xxx11	xxx0	–	予測不能	–	–

a. マルチプロセッシング拡張機能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter A6
Thumb Instruction Set Encoding

This chapter introduces the Thumb instruction set and describes how it uses the ARM programmers' model. It contains the following sections:

- *Thumb instruction set encoding* on page A6-2
- *16-bit Thumb instruction encoding* on page A6-6
- *32-bit Thumb instruction encoding* on page A6-14.

For details of the differences between the Thumb and ThumbEE instruction sets see Chapter A9 *ThumbEE*.

Note

- Architecture variant information in this chapter describes the architecture variant or extension in which the instruction encoding was introduced into the Thumb instruction set.
- In the decode tables in this chapter, an entry of - for a field value means the value of the field does not affect the decoding.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 A6 章
Thumb 命令セットのエンコード

本章では、Thumb 命令セットと、この命令セットでの ARM プログラマモデルの使用方法について説明します。本章は次のセクションから構成されています。

- *Thumb 命令セットのエンコード* P. A6-2
- *16 ビット Thumb 命令のエンコード* P. A6-6
- *32 ビット Thumb 命令のエンコード* P. A6-14

Thumb 命令セットと ThumbEE 命令セットとの違いの詳細については、第 A9 章「*ThumbEE*」を参照して下さい。

注

- 本章のアーキテクチャバリアント情報では、その命令エンコードが、どのアーキテクチャのバリアントや拡張機能で Thumb 命令セットに導入されたかが記載されています。
- 本章のデコード表においてフィールド値に「-」が記入されているのは、フィールドの値がデコードに影響を与えないことを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.1 Thumb instruction set encoding

The Thumb instruction stream is a sequence of halfword-aligned halfwords. Each Thumb instruction is either a single 16-bit halfword in that stream, or a 32-bit instruction consisting of two consecutive halfwords in that stream.

If bits [15:11] of the halfword being decoded take any of the following values, the halfword is the first halfword of a 32-bit instruction:

- 0b11101
- 0b11110
- 0b11111.

Otherwise, the halfword is a 16-bit instruction.

For details of the encoding of 16-bit Thumb instructions see *16-bit Thumb instruction encoding* on page A6-6.

For details of the encoding of 32-bit Thumb instructions see *32-bit Thumb instruction encoding* on page A6-14.

A6.1.1 UNDEFINED and UNPREDICTABLE instruction set space

An attempt to execute an unallocated instruction results in either:

- Unpredictable behavior. The instruction is described as UNPREDICTABLE.
- An Undefined Instruction exception. The instruction is described as UNDEFINED.

An instruction is UNDEFINED if it is declared as UNDEFINED in an instruction description, or in this chapter.

An instruction is UNPREDICTABLE if:

- a bit marked (0) or (1) in the encoding diagram of an instruction is not 0 or 1 respectively, and the pseudocode for that encoding does not indicate that a different special case applies
- it is declared as UNPREDICTABLE in an instruction description or in this chapter.

Unless otherwise specified:

- Thumb instructions introduced in an architecture variant are either UNPREDICTABLE or UNDEFINED in earlier architecture variants.
- A Thumb instruction that is provided by one or more of the architecture extensions is either UNPREDICTABLE or UNDEFINED in an implementation that does not include any of those extensions.

In both cases, the instruction is UNPREDICTABLE if it is a 32-bit instruction in an architecture variant before ARMv6T2, and UNDEFINED otherwise.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.1 Thumb 命令セットのエンコード

Thumb 命令ストリームは、ハーフワードでアラインされたハーフワードのシーケンスです。そのストリーム内の各 Thumb 命令は、単一の 16 ビット命令か、2 つの連続するハーフワードで構成された 32 ビット命令のどちらかです。

デコードされるハーフワードのビット [15:11] が、次のいずれかの値であれば、そのハーフワードは 32 ビット命令の最初のハーフワードです。

- 0b11101
- 0b11110
- 0b11111

それ以外の場合、そのハーフワードは 16 ビット命令です。

16 ビット Thumb 命令のエンコードの詳細については、*16 ビット Thumb 命令のエンコード* P. A6- 6 を参照して下さい。

32 ビット Thumb 命令のエンコードの詳細については、*32 ビット Thumb 命令のエンコード* P. A6- 14 を参照して下さい。

A6.1.1 未定義および予測不能の命令セット空間

未割り当て命令の実行を試みると、次のいずれかの結果が発生します。

- 予測不能な動作。このような命令は、予測不能と記述されています。
- 未定義命令例外。このような命令は、未定義と記述されています。

命令の説明または本章の中で未定義と記載されている命令は、未定義です。

次のいずれかに該当する命令は予測不能です。

- 命令のエンコード図で (0) または (1) と表記されたビットがそれぞれ 0 または 1 ではなく、そのエンコードの擬似コードで、別の特別な条件が適用されることが示されていない。
- 命令の説明または本章で、予測不能と記載されている。

他に特に規定がない場合、次の条件が適用されます。

- あるアーキテクチャバリアントで導入された Thumb 命令は、それ以前のアーキテクチャバリアントでは予測不能または未定義です。
- 1 つ以上のアーキテクチャ拡張機能で提供されている Thumb 命令は、その拡張機能を含まない実装では予測不能または未定義です。

両方のケースにおいて、その命令が ARMv6T2 以前のアーキテクチャバリアントにおける 32 ビット命令の場合は予測不能で、それ以外の場合は未定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.1.2 Use of 0b1111 as a register specifier

The use of 0b1111 as a register specifier is not normally permitted in Thumb instructions. When a value of 0b1111 is permitted, a variety of meanings is possible. For register reads, these meanings are:

- Read the PC value, that is, the address of the current instruction + 4. The base register of the table branch instructions TBB and TBH can be the PC. This enables branch tables to be placed in memory immediately after the instruction.

  Note

  Use of the PC as the base register in the STC instruction is deprecated in ARMv7.

- Read the word-aligned PC value, that is, the address of the current instruction + 4, with bits [1:0] forced to zero. The base register of LDC, LDR, LDRB, LDRD (pre-indexed, no writeback), LDRH, LDRSB, and LDRSH instructions can be the word-aligned PC. This enables PC-relative data addressing. In addition, some encodings of the ADD and SUB instructions permit their source registers to be 0b1111 for the same purpose.
- Read zero. This is done in some cases when one instruction is a special case of another, more general instruction, but with one operand zero. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.

For register writes, these meanings are:

- The PC can be specified as the destination register of an LDR instruction. This is done by encoding Rt as 0b1111. The loaded value is treated as an address, and the effect of execution is a branch to that address. bit [0] of the loaded value selects whether to execute ARM or Thumb instructions after the branch.

  Some other instructions write the PC in similar ways, either implicitly (for example branch instructions) or by using a register mask rather than a register specifier (LDM). The address to branch to can be:

  — a loaded value, for example, RFE
  — a register value, for example, BX
  — the result of a calculation, for example, TBB or TBH.

  The method of choosing the instruction set used after the branch can be:

  — similar to the LDR case, for LDM or BX
  — a fixed instruction set other than the one currently being used, for example, the immediate form of BLX
  — unchanged, for example branch instructions
  — set from the (J,T) bits of the SPSR, for RFE and SUBS PC,LR,#imm8.

- Discard the result of a calculation. This is done in some cases when one instruction is a special case of another, more general instruction, but with the result discarded. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.1.2 レジスタ指定子として 0b1111 を使用する場合

通常、Thumb 命令ではレジスタ指定子として 0b1111 を使用することはできません。値として 0b1111 が許可される場合、各種の意味に使用される可能性があります。レジスタ読み出しの場合は、次のような意味になります。

- PC の値、つまり現在の命令 + 4 のアドレスを読み出す。テーブル分岐命令 TBB および TBH では、ベースレジスタを PC にすることができます。これによって、メモリ内で分岐テーブルを命令の直後に配置することができます。

  注

  STC 命令でベースレジスタとして PC を使用することは、ARMv7 では推奨されません。

- ワードアラインドの PC の値を読み出す。これは、現在の命令 + 4 のアドレスを読み出し、その値のビット [1:0] を強制的に 0 に設定することを意味します。LDC、LDR、LDRB、LDRD（プリインデクス、ライトバックなし）、LDRH、LDRSB、LDRSH の各命令では、ワードアラインドの PC をベースレジスタにすることができます。これによって、PC 相対のデータアドレシングが可能になります。さらに、ADD 命令および SUB 命令の一部のエンコードでは、同じ目的でソースレジスタを 0b1111 にすることができます。
- 0 を読み出す。この動作は、その命令が、別のより汎用的な命令の特殊なケースで、1 つのオペランドが 0 である一部の場合に発生します。この場合、その命令の説明は別のページに記載され、より汎用的な命令の説明において、擬似コードの特殊なケースとしてそのページへの参照が示されています。

レジスタ書き込みの場合は、次のような意味になります。

- LDR 命令のデスティネーションレジスタとして PC を指定できます。この場合、Rt が 0b1111 としてエンコードされます。ロードした値はアドレスとして扱われ、実行の結果そのアドレスへの分岐が行われます。ロードされた値のビット [0] により、分岐の後で ARM 命令と Thumb 命令のどちらを実行するかが選択されます。

  他の一部の命令では、暗黙的に（分岐命令など）、またはレジスタ指定子の代わりにレジスタマスクを使用して（LDM）、同様に PC に書き込むことができます。分岐先のアドレスは次のいずれかです。

  – ロードされた値（RFE など）
  – レジスタの値（BX など）
  – 計算の結果（TBB や TBH など）

  分岐の後で使用される命令セットは、次のいずれかの方法で選択されます。

  – LDM または BX については、LDR と同様の方法で選択されます。
  – BLX のイミディエート形式などでは、現在使用されている命令セットとは異なる所定の命令セットが使用されます。
  – 分岐命令などでは、変更されません。
  – RFE および SUBS PC,LR,#imm8 については、SPSR の (J,T) ビットから設定されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- If the destination register specifier of an LDRB, LDRH, LDRSB, or LDRSH instruction is 0b1111, the instruction is a memory hint instead of a load operation.
- If the destination register specifier of an MRC instruction is 0b1111, bits [31:28] of the value transferred from the coprocessor are written to the N, Z, C, and V flags in the APSR, and bits [27:0] are discarded.

A6.1.3 Use of 0b1101 as a register specifier

R13 is defined in the Thumb instruction set so that its use is primarily as a stack pointer, and R13 is normally identified as SP in Thumb instructions. In 32-bit Thumb instructions, if you use R13 as a general-purpose register beyond the architecturally defined constraints described in this section, the results are UNPREDICTABLE.

The restrictions applicable to R13 are described in:

- *R13[1:0] definition*
- *32-bit Thumb instruction support for R13*.

See also *16-bit Thumb instruction support for R13* on page A6-5.

R13[1:0] definition

Bits [1:0] of R13 are SBZP. Writing a nonzero value to bits [1:0] causes UNPREDICTABLE behavior.

32-bit Thumb instruction support for R13

R13 instruction support is restricted to the following:

- R13 as the source or destination register of a MOV instruction. Only register to register transfers without shifts are supported, with no flag setting:

```
MOV     SP,<Rm>
MOV     <Rn>,SP
```

- Using the following instructions to adjust R13 up or down by a multiple of 4:

```
ADD{W}  SP,SP,#<imm>
SUB{W}  SP,SP,#<imm>
ADD     SP,SP,<Rm>
ADD     SP,SP,<Rm>,LSL #<n>      ; For <n> = 1,2,3
SUB     SP,SP,<Rm>
SUB     SP,SP,<Rm>,LSL #<n>      ; For <n> = 1,2,3
```

- R13 as a base register <Rn> of any load/store instruction. This supports SP-based addressing for load, store, or memory hint instructions, with positive or negative offsets, with and without writeback.
- R13 as the first operand <Rn> in any ADD{S}, CMN, CMP, or SUB{S} instruction. The add and subtract instructions support SP-based address generation, with the address going into a general-purpose register. CMN and CMP are useful for stack checking in some circumstances.
- R13 as the transferred register <Rt> in any LDR or STR instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- 計算の結果を破棄する。この動作は、その命令が、別のより汎用的な命令の特殊なケースで、結果が破棄される場合に発生します。この場合、その命令の説明は別のページに記載され、より汎用的な命令の説明において、擬似コードの特殊なケースとしてそのページへの参照が示されています。
- LDRB、LDRH、LDRSB、LDRSH 命令のデスティネーションレジスタ指定子が 0b1111 の場合、その命令はロード操作ではなくメモリヒントです。
- MRC 命令のデスティネーションレジスタ指定子が 0b1111 の場合は、コプロセッサから転送された値のビット [31:28] が APSR の N、Z、C、V フラグに書き込まれ、ビット [27:0] は破棄されます。

A6.1.3 レジスタ指定子として 0b1101 を使用する場合

R13 は、主にスタックポインタとして使用するように Thumb 命令セット内で定義されており、Thumb 命令では一般に SP と表記されます。32 ビット Thumb 命令で、このセクションに記載されているアーキテクチャ定義の制限を無視して R13 を汎用レジスタとして使用した場合、結果は予測不能です。

R13 に適用される制限については、次のセクションを参照して下さい。

- *R13[1:0] の定義*
- *32 ビット Thumb 命令での R13 のサポート*

16 ビット Thumb 命令での R13 のサポート P. A6-5 も参照して下さい。

R13[1:0] の定義

R13 のビット [1:0] は SBZP です。ビット [1:0] に 0 以外の値を書き込んだ場合、動作は予測不能となります。

32 ビット Thumb 命令での R13 のサポート

R13 命令のサポートは、次の操作に制限されます。

- MOV 命令のソースレジスタまたはデスティネーションレジスタとして R13 を使用する。シフトを伴わないレジスタ間転送のみサポートされ、フラグはセットされません。

```
MOV     SP,<Rm>
MOV     <Rn>,SP
```

- 次の命令を使用して、R13 を 4 の倍数ずつ上下に調整する。

```
ADD{W}  SP,SP,#<imm>
SUB{W}  SP,SP,#<imm>
ADD     SP,SP,<Rm>
ADD     SP,SP,<Rm>,LSL #<n>      ; For <n> = 1,2,3
SUB     SP,SP,<Rm>
SUB     SP,SP,<Rm>,LSL #<n>      ; For <n> = 1,2,3
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

16-bit Thumb instruction support for R13

For 16-bit data-processing instructions that affect high registers, R13 can only be used as described in *32-bit Thumb instruction support for R13* on page A6-4. Any other use is deprecated. This affects the high register forms of `CMP` and `ADD`, where the use of R13 as `<Rm>` is deprecated.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- 任意のロード / ストア命令のベースレジスタ `<Rn>` として R13 を使用する。これによって、ロード命令、ストア命令、メモリヒント命令で、SP をベースとしたアドレシング（正または負のオフセット、ライトバックありまたはなし）がサポートされます。
- 任意の `ADD{S}`、`CMN`、`CMP`、`SUB{S}` 命令で、最初のオペランド `<Rn>` として R13 を使用する。加算命令と減算命令では、SP をベースとしたアドレスを生成し、そのアドレスを汎用レジスタへ転送する操作がサポートされます。`CMN` および `CMP` は、特定の状況でスタックのチェックを行うのに便利です。
- 任意の `LDR` 命令または `STR` 命令で、転送先レジスタ `<Rt>` として R13 を使用する。

16 ビット Thumb 命令での R13 のサポート

上位レジスタに影響を及ぼす 16 ビットデータ処理命令の場合、*32 ビット Thumb 命令での R13 のサポート* P. A6-4 で説明されている方法でのみ R13 を使用できます。それ以外の使用は推奨されません。これは `CMP` と `ADD` の上位レジスタ形式に影響し、R13 を `<Rm>` として使用することは推奨されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.2 16-bit Thumb instruction encoding

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
Opcode	

Table A6-1 shows the allocation of 16-bit instruction encodings.

Table A6-1 16-bit Thumb instruction encoding

Opcode	Instruction or instruction class	Variant
00xxxx	*Shift (immediate), add, subtract, move, and compare* on page A6-7	-
010000	*Data-processing* on page A6-8	-
010001	*Special data instructions and branch and exchange* on page A6-9	-
01001x	Load from Literal Pool, see *LDR (literal)* on page A8-122	v4T
0101xx	*Load/store single data item* on page A6-10	-
011xxx		
100xxx		
10100x	Generate PC-relative address, see *ADR* on page A8-32	v4T
10101x	Generate SP-relative address, see *ADD (SP plus immediate)* on page A8-28	v4T
1011xx	*Miscellaneous 16-bit instructions* on page A6-11	-
11000x	Store multiple registers, see *STM / STMIA / STMEA* on page A8-374 [a]	v4T
11001x	Load multiple registers, see *LDM / LDMIA / LDMFD* on page A8-110 [a]	v4T
1101xx	*Conditional branch, and Supervisor Call* on page A6-13	-
11100x	Unconditional Branch, see *B* on page A8-44	v4T

a. In ThumbEE, 16-bit load/store multiple instructions are not available. This encoding is used for special ThumbEE instructions. For details see Chapter A9 *ThumbEE*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A6.2 16 ビット Thumb 命令のエンコード

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
オペコード	

16 ビット命令のエンコードの割り当てを、表 A6-1 に示します。

表 A6-1 16 ビット Thumb 命令のエンコード

オペコード	命令または命令クラス	バリアント
00xxxx	*シフト（イミディエート）、加算、減算、移動、比較* P. A6-7	–
010000	*データ処理* P. A6-8	–
010001	*特殊なデータ命令および分岐と状態遷移* P. A6-9	–
01001x	リテラルプールからのロード。*LDR（リテラル）* P. A8-122 参照	v4T
0101xx	*単一のデータ項目のロード / ストア* P. A6-10	–
011xxx		
100xxx		
10100x	PC 相対アドレスの生成。*ADR* P. A8-32 参照	v4T
10101x	SP 相対アドレスの生成。*ADD（SP + イミディエート）* P. A8-28 参照	v4T
1011xx	*その他の 16 ビット命令* P. A6-11	–
11000x	複数のレジスタのストア。*STM*、*STMIA*、*STMEA* P. A8-374 参照[a]	v4T
11001x	複数のレジスタのロード。*LDM/LDMIA/LDMFD* P. A8-110 参照 [a]	v4T
1101xx	*条件付き分岐およびスーパバイザコール* P. A6-13	–
11100x	無条件分岐。*B* P. A8-44 参照	v4T

a. ThumbEE では、16 ビットの複数ロード / ストア命令を使用できません。このエンコードは、特殊な ThumbEE 命令で使用されます。詳細については、第 A9 章「*ThumbEE*」を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.2.1 Shift (immediate), add, subtract, move, and compare

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	Opcode													

Table A6-2 shows the allocation of encodings in this space.

All these instructions are available since the Thumb instruction set was introduced in ARMv4T.

Table A6-2 16-bit Thumb shift (immediate), add, subtract, move, and compare instructions

Opcode	Instruction	See
000xx	Logical Shift Left	*LSL (immediate)* on page A8-178
001xx	Logical Shift Right	*LSR (immediate)* on page A8-182
010xx	Arithmetic Shift Right	*ASR (immediate)* on page A8-40
01100	Add register	*ADD (register)* on page A8-24
01101	Subtract register	*SUB (register)* on page A8-422
01110	Add 3-bit immediate	*ADD (immediate, Thumb)* on page A8-20
01111	Subtract 3-bit immediate	*SUB (immediate, Thumb)* on page A8-418
100xx	Move	*MOV (immediate)* on page A8-194
101xx	Compare	*CMP (immediate)* on page A8-80
110xx	Add 8-bit immediate	*ADD (immediate, Thumb)* on page A8-20
111xx	Subtract 8-bit immediate	*SUB (immediate, Thumb)* on page A8-418

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.2.1 シフト（イミディエート）、加算、減算、移動、比較

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	オペコード													

この空間でのエンコードの割り当てを、表 A6-2 に示します。

ARMv4T で Thumb 命令セットが導入されて以来、これらの命令はすべて使用できます。

表 A6-2 16 ビットの Thumb シフト（イミディエート）、加算、減算、移動、比較命令

オペコード	命令	参照先
000xx	論理左シフト	*LSL（イミディエート）* P. A8-178
001xx	論理右シフト	*LSR（イミディエート）* P. A8-182
010xx	算術右シフト	*ASR（イミディエート）* P. A8-40
01100	レジスタの加算	*ADD（レジスタ）* P. A8-24
01101	レジスタの減算	*SUB（レジスタ）* P. A8-422
01110	3 ビットイミディエート値の加算	*ADD（イミディエート、Thumb）* P. A8-20
01111	3 ビットイミディエート値の減算	*SUB（イミディエート、Thumb）* P. A8-418
100xx	移動	*MOV（イミディエート）* P. A8-194
101xx	比較	*CMP（イミディエート）* P. A8-80
110xx	8 ビットイミディエート値の加算	*ADD（イミディエート、Thumb）* P. A8-20
111xx	8 ビットイミディエート値の減算	*SUB（イミディエート、Thumb）* P. A8-418

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.2.2 Data-processing

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	Opcode									

Table A6-3 shows the allocation of encodings in this space.

All these instructions are available since the Thumb instruction set was introduced in ARMv4T.

Table A6-3 16-bit Thumb data-processing instructions

Opcode	Instruction	See
0000	Bitwise AND	*AND (register)* on page A8-36
0001	Bitwise Exclusive OR	*EOR (register)* on page A8-96
0010	Logical Shift Left	*LSL (register)* on page A8-180
0011	Logical Shift Right	*LSR (register)* on page A8-184
0100	Arithmetic Shift Right	*ASR (register)* on page A8-42
0101	Add with Carry	*ADC (register)* on page A8-16
0110	Subtract with Carry	*SBC (register)* on page A8-304
0111	Rotate Right	*ROR (register)* on page A8-280
1000	Test	*TST (register)* on page A8-456
1001	Reverse Subtract from 0	*RSB (immediate)* on page A8-284
1010	Compare High Registers	*CMP (register)* on page A8-82
1011	Compare Negative	*CMN (register)* on page A8-76
1100	Bitwise OR	*ORR (register)* on page A8-230
1101	Multiply Two Registers	*MUL* on page A8-212
1110	Bitwise Bit Clear	*BIC (register)* on page A8-52
1111	Bitwise NOT	*MVN (register)* on page A8-216

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.2.2 データ処理

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	オペコード									

この空間でのエンコードの割り当てを、表 A6-3 に示します。

ARMv4T で Thumb 命令セットが導入されて以来、これらの命令はすべて使用できます。

表 A6-3 16 ビット Thumb データ処理命令

オペコード	命令	参照先
0000	ビット単位論理積	*AND（レジスタ）* P. A8-36
0001	ビット単位排他的論理和	*EOR（レジスタ）* P. A8-96
0010	論理左シフト	*LSL（レジスタ）* P. A8-180
0011	論理右シフト	*LSR（レジスタ）* P. A8-184
0100	算術右シフト	*ASR（レジスタ）* P. A8-42
0101	キャリー付き加算	*ADC（レジスタ）* P. A8-16
0110	キャリー付き減算	*SBC（レジスタ）* P. A8-304
0111	右ローテート	*ROR（レジスタ）* P. A8-280
1000	テスト	*TST（レジスタ）* P. A8-456
1001	0 からの逆減算	*RSB（イミディエート）* P. A8-284
1010	上位レジスタの比較	*CMP（レジスタ）* P. A8-82
1011	比較否定	*CMN（レジスタ）* P. A8-76
1100	ビット単位論理和	*ORR（レジスタ）* P. A8-230
1101	2 つのレジスタの乗算	*MUL* P. A8-212
1110	ビット単位のビットクリア	*BIC（レジスタ）* P. A8-52
1111	ビット単位否定	*MVN（レジスタ）* P. A8-216

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.2.3 Special data instructions and branch and exchange

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	Opcode									

Table A6-4 shows the allocation of encodings in this space.

Table A6-4 16-bit Thumb special data instructions and branch and exchange

Opcode	Instruction	See	Variant
0000	Add Low Registers	*ADD (register)* on page A8-24	v6T2 [a]
0001 001x	Add High Registers	*ADD (register)* on page A8-24	v4T
0100	UNPREDICTABLE	-	-
0101 011x	Compare High Registers	*CMP (register)* on page A8-82	v4T
1000	Move Low Registers	*MOV (register)* on page A8-196	v6 [a]
1001 101x	Move High Registers	*MOV (register)* on page A8-196	v4T
110x	Branch and Exchange	*BX* on page A8-62	v4T
111x	Branch with Link and Exchange	*BLX (register)* on page A8-60	v5T [a]

a. UNPREDICTABLE in earlier variants.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A6.2.3 特殊なデータ命令および分岐と状態遷移

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	オペコード									

この空間でのエンコードの割り当てを、表 A6-4 に示します。

表 A6-4 16 ビット Thumb の特殊なデータ命令および分岐と状態遷移

オペコード	命令	参照先	バリアント
0000	下位レジスタの加算	*ADD（レジスタ）* P. A8-24	v6T2 [a]
0001 001x	上位レジスタの加算	*ADD（レジスタ）* P. A8-24	v4T
0100	予測不能	–	–
0101 011x	上位レジスタの比較	*CMP（レジスタ）* P. A8-82	v4T
1000	下位レジスタの移動	*MOV（レジスタ）* P. A8-196	v6 [a]
1001 101x	上位レジスタの移動	*MOV（レジスタ）* P. A8-196	v4T
110x	分岐と状態遷移	*BX* P. A8-62	v4T
111x	リンク付き分岐と状態遷移	*BLX（レジスタ）* P. A8-60	v5T [a]

a. それ以前のバリアントでは予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.2.4 Load/store single data item

15 14 13 12	11 10	9 8 7 6 5 4 3 2 1 0
opA	opB	

These instructions have one of the following values in opA:

- 0b0101
- 0b011x
- 0b100x.

Table A6-5 shows the allocation of encodings in this space.

All these instructions are available since the Thumb instruction set was introduced in ARMv4T.

Table A6-5 16-bit Thumb Load/store instructions

opA	opB	Instruction	See
0101	000	Store Register	*STR (register)* on page A8-386
	001	Store Register Halfword	*STRH (register)* on page A8-412
	010	Store Register Byte	*STRB (register)* on page A8-392
	011	Load Register Signed Byte	*LDRSB (register)* on page A8-164
	100	Load Register	*LDR (register)* on page A8-124
	101	Load Register Halfword	*LDRH (register)* on page A8-156
	110	Load Register Byte	*LDRB (register)* on page A8-132
	111	Load Register Signed Halfword	*LDRSH (register)* on page A8-172
0110	0xx	Store Register	*STR (immediate, Thumb)* on page A8-382
	1xx	Load Register	*LDR (immediate, Thumb)* on page A8-118
0111	0xx	Store Register Byte	*STRB (immediate, Thumb)* on page A8-388
	1xx	Load Register Byte	*LDRB (immediate, Thumb)* on page A8-126
1000	0xx	Store Register Halfword	*STRH (immediate, Thumb)* on page A8-408
	1xx	Load Register Halfword	*LDRH (immediate, Thumb)* on page A8-150
1001	0xx	Store Register SP relative	*STR (immediate, Thumb)* on page A8-382
	1xx	Load Register SP relative	*LDR (immediate, Thumb)* on page A8-118

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.2.4 単一のデータ項目のロード / ストア

15 14 13 12	11 10	9 8 7 6 5 4 3 2 1 0
opA	opB	

これらの命令の opA は、次のいずれかの値です。

- 0b0101
- 0b011x
- 0b100x

この空間でのエンコードの割り当てを、表 A6-5 に示します。

ARMv4T で Thumb 命令セットが導入されて以来、これらの命令はすべて使用できます。

表 A6-5 16 ビット Thumb ロード / ストア命令

opA	opB	命令	参照先
0101	000	レジスタストア	*STR（レジスタ）* P. A8-386
	001	レジスタストア ハーフワード	*STRH（レジスタ）* P. A8-412
	010	レジスタストア バイト	*STRB（レジスタ）* P. A8-392
	011	レジスタロード符号付きバイト	*LDRSB（レジスタ）* P. A8-164
	100	レジスタロード	*LDR（レジスタ）* P. A8-124
	101	レジスタロード ハーフワード	*LDRH（レジスタ）* P. A8-156
	110	レジスタロード バイト	*LDRB（レジスタ）* P. A8-132
	111	レジスタロード符号付き ハーフワード	*LDRSH（レジスタ）* P. A8-172
0110	0xx	レジスタストア	*STR（イミディエート、Thumb）* P. A8-382
	1xx	レジスタロード	*LDR（イミディエート、Thumb）* P. A8-118
0111	0xx	レジスタストア バイト	*STRB（イミディエート、Thumb）* P. A8-388
	1xx	レジスタロード バイト	*LDRB（イミディエート、Thumb）* P. A8-126
1000	0xx	レジスタストア ハーフワード	*STRH（イミディエート、Thumb）* P. A8-408
	1xx	レジスタロード ハーフワード	*LDRH（イミディエート、Thumb）* P. A8-150
1001	0xx	レジスタストア SP 相対	*STR（イミディエート、Thumb）* P. A8-382
	1xx	レジスタロード SP 相対	*LDR（イミディエート、Thumb）* P. A8-118

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.2.5 Miscellaneous 16-bit instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	Opcode											

Table A6-6 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A6-6 Miscellaneous 16-bit instructions

Opcode	Instruction	See	Variant
0110010	Set Endianness	*SETEND* on page A8-314	v6
0110011	Change Processor State	*CPS* on page B6-3	v6
00000xx	Add Immediate to SP	*ADD (SP plus immediate)* on page A8-28	v4T
00001xx	Subtract Immediate from SP	*SUB (SP minus immediate)* on page A8-426	v4T
0001xxx	Compare and Branch on Zero	*CBNZ, CBZ* on page A8-66	v6T2
001000x	Signed Extend Halfword	*SXTH* on page A8-444	v6
001001x	Signed Extend Byte	*SXTB* on page A8-440	v6
001010x	Unsigned Extend Halfword	*UXTH* on page A8-524	v6
001011x	Unsigned Extend Byte	*UXTB* on page A8-520	v6
0011xxx	Compare and Branch on Zero	*CBNZ, CBZ* on page A8-66	v6T2
010xxxx	Push Multiple Registers	*PUSH* on page A8-248	v4T
1001xxx	Compare and Branch on Nonzero	*CBNZ, CBZ* on page A8-66	v6T2
101000x	Byte-Reverse Word	*REV* on page A8-272	v6
101001x	Byte-Reverse Packed Halfword	*REV16* on page A8-274	v6
101011x	Byte-Reverse Signed Halfword	*REVSH* on page A8-276	v6
1011xxx	Compare and Branch on Nonzero	*CBNZ, CBZ* on page A8-66	v6T2
110xxxx	Pop Multiple Registers	*POP* on page A8-246	v4T
1110xxx	Breakpoint	*BKPT* on page A8-56	v5
1111xxx	If-Then, and hints	*If-Then, and hints* on page A6-12	-

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A6.2.5 その他の 16 ビット命令

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	オペコード											

この空間でのエンコードの割り当てを、表 A6-6 に示します。この空間に含まれる他のエンコードは未定義です。

表 A6-6 その他の 16 ビット命令

オペコード	命令	参照先	バリアント
0110010	エンディアン形式の設定	*SETEND* P. A8-314	v6
0110011	プロセッサ状態変更	*CPS* P. B6-3	v6
00000xx	SP へのイミディエート値の加算	*ADD（SP + イミディエート）* P. A8-28	v4T
00001xx	SP からのイミディエート値の減算	*SUB（SP − イミディエート）* P. A8-426	v4T
0001xxx	比較して 0 で分岐	*CBNZ*、*CBZ* P. A8-66	v6T2
001000x	符号付きハーフワード拡張	*SXTH* P. A8-444	v6
001001x	符号付きバイト拡張	*SXTB* P. A8-440	v6
001010x	符号なしハーフワード拡張	*UXTH* P. A8-524	v6
001011x	符号なしバイト拡張	*UXTB* P. A8-520	v6
0011xxx	比較して 0 で分岐	*CBNZ*、*CBZ* P. A8-66	v6T2
010xxxx	複数レジスタプッシュ	*PUSH* P. A8-248	v4T
1001xxx	比較して非 0 で分岐	*CBNZ*、*CBZ* P. A8-66	v6T2
101000x	ワードのバイト反転	*REV* P. A8-272	v6
101001x	パッキングされたハーフワードのバイト反転	*REV16* P. A8-274	v6
101011x	符号付きハーフワードのバイト反転	*REVSH* P. A8-276	v6
1011xxx	比較して非 0 で分岐	*CBNZ*、*CBZ* P. A8-66	v6T2
110xxxx	複数レジスタポップ	*POP* P. A8-246	v4T
1110xxx	ブレークポイント	*BKPT* P. A8-56	v5
1111xxx	If-Then とヒント	*If-Then とヒント* P. A6-12	−

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If-Then, and hints

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	opA				opB			

Table A6-7 shows the allocation of encodings in this space.

Other encodings in this space are unallocated hints. They execute as NOPs, but software must not use them.

Table A6-7 Miscellaneous 16-bit instructions

opA	opB	Instruction	See	Variant
-	not 0000	If-Then	*IT* on page A8-104	v6T2
0000	0000	No Operation hint	*NOP* on page A8-222	v6T2
0001	0000	Yield hint	*YIELD* on page A8-812	v7
0010	0000	Wait For Event hint	*WFE* on page A8-808	v7
0011	0000	Wait For Interrupt hint	*WFI* on page A8-810	v7
0100	0000	Send Event hint	*SEV* on page A8-316	v7

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If-Then とヒント

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	opA				opB			

この空間でのエンコードの割り当てを、表 A6-7 に示します。

この空間に含まれる他のエンコードは未割り当てのヒントです。NOP として実行されますが、ソフトウェアでは使用しないで下さい。

表 A6-7 その他の 16 ビット命令

opA	opB	命令	参照先	バリアント
-	0000 以外	If-Then	*IT* P. A8-104	v6T2
0000	0000	無操作ヒント	*NOP* P. A8-222	v6T2
0001	0000	イールドヒント	*YIELD* P. A8-812	v7
0010	0000	イベント待ちヒント	*WFE* P. A8-808	v7
0011	0000	割り込み待ちヒント	*WFI* P. A8-810	v7
0100	0000	イベント送信ヒント	*SEV* P. A8-316	v7

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.2.6 Conditional branch, and Supervisor Call

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Opcode											

Table A6-8 shows the allocation of encodings in this space.

All these instructions are available since the Thumb instruction set was introduced in ARMv4T.

Table A6-8 Conditional branch and Supervisor Call instructions

Opcode	Instruction	See
not 111x	Conditional branch	*B* on page A8-44
1110	Permanently UNDEFINED. This space will not be allocated in future.	
1111	Supervisor Call	*SVC (previously SWI)* on page A8-430

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.2.6 条件付き分岐およびスーパバイザコール

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	オペコード											

この空間でのエンコードの割り当てを、表 A6-8 に示します。

ARMv4T で Thumb 命令セットが導入されて以来、これらの命令はすべて使用できます。

表 A6-8 条件付き分岐およびスーパバイザコール命令

オペコード	命令	参照先
111x 以外	条件付き分岐	*B* P. A8-44
1110	恒久的に未定義。この空間は将来も割り当てられません。	
1111	スーパバイザコール	*SVC（以前の SWI）* P. A8-430

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3 32-bit Thumb instruction encoding

15 14 13	12 11 10 9 8 7 6 5 4 3 2 1 0		15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1	op1	op2	op	

If op1 == 0b00, a 16-bit instruction is encoded, see *16-bit Thumb instruction encoding* on page A6-6.

Table A6-9 shows the allocation of encodings in this space.

Table A6-9 32-bit Thumb instruction encoding

op1	op2	op	Instruction class, see
01	00xx0xx	-	*Load/store multiple* on page A6-23
	00xx1xx	-	*Load/store dual, load/store exclusive, table branch* on page A6-24
	01xxxxx	-	*Data-processing (shifted register)* on page A6-31
	1xxxxxx	-	*Coprocessor instructions* on page A6-40
10	x0xxxxx	0	*Data-processing (modified immediate)* on page A6-15
	x1xxxxx	0	*Data-processing (plain binary immediate)* on page A6-19
	-	1	*Branches and miscellaneous control* on page A6-20
11	000xxx0	-	*Store single data item* on page A6-30
	001xxx0	-	*Advanced SIMD element or structure load/store instructions* on page A7-27
	00xx001	-	*Load byte, memory hints* on page A6-28
	00xx011	-	*Load halfword, memory hints* on page A6-26
	00xx101	-	*Load word* on page A6-25
	00xx111	-	UNDEFINED
	010xxxx	-	*Data-processing (register)* on page A6-33
	0110xxx	-	*Multiply, multiply accumulate, and absolute difference* on page A6-38
	0111xxx	-	*Long multiply, long multiply accumulate, and divide* on page A6-39
	1xxxxxx	-	*Coprocessor instructions* on page A6-40

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A6.3 32 ビット Thumb 命令のエンコード

15 14 13	12 11 10 9 8 7 6 5 4 3 2 1 0		15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1	op1	op2	op	

op1 == 0b00 の場合は、16 ビット命令がエンコードされます。*16 ビット Thumb 命令のエンコード* P. A6-6 を参照して下さい。

この空間でのエンコードの割り当てを、表 A6-9 に示します。

表 A6-9 32 ビット Thumb 命令のエンコード

op1	op2	op	命令クラス（参照先）
01	00xx0xx	–	*複数ロード / ストア* P. A6-23
	00xx1xx	–	*デュアルロード / ストア、排他ロード / ストア、テーブル分岐* P. A6-24
	01xxxxx	–	*データ処理（シフトしたレジスタ）* P. A6-31
	1xxxxxx	–	*コプロセッサ命令* P. A6-40
10	x0xxxxx	0	*データ処理（修飾イミディエート）* P. A6-15
	x1xxxxx	0	*データ処理（普通のバイナリイミディエート）* P. A6-19
	–	1	*分岐およびその他の制御* P. A6-20
11	000xxx0	–	*単一データ項目のストア* P. A6-30
	001xxx0	–	*アドバンスト SIMD のエレメントまたは構造ロード / ストア命令* P. A7-27
	00xx001	–	*バイトロード、メモリヒント* P. A6-28
	00xx011	–	*ハーフワードロード、メモリヒント* P. A6-26
	00xx101	–	*ワードロード* P. A6-25
	00xx111	–	未定義
	010xxxx	–	*データ処理（レジスタ）* P. A6-33
	0110xxx	–	*乗算、積和演算、および絶対値の差* P. A6-38
	0111xxx	–	*ロング乗算、ロング積和演算、および除算* P. A6-39
	1xxxxxx	–	*コプロセッサ命令* P. A6-40

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.1 Data-processing (modified immediate)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0		0	op				S	Rn				0				Rd											

Table A6-10 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

In the Rn, Rd and S columns, - indicates that the field value of the field does affect the decoding.

These encodings are all available in ARMv6T2 and above.

Table A6-10 32-bit modified immediate data-processing instructions

op	Rn	Rd	S	Instruction	See
0000	-	not 1111	x	Bitwise AND	*AND (immediate)* on page A8-34
	-	1111	0	UNPREDICTABLE	-
	-	1111	1	Test	*TST (immediate)* on page A8-454
0001	-	-	-	Bitwise Bit Clear	*BIC (immediate)* on page A8-50
0010	not 1111	-	-	Bitwise OR	*ORR (immediate)* on page A8-228
	1111	-	-	Move	*MOV (immediate)* on page A8-194
0011	not 1111	-	-	Bitwise OR NOT	*ORN (immediate)* on page A8-224
	1111	-	-	Bitwise NOT	*MVN (immediate)* on page A8-214
0100	-	not 1111	x	Bitwise Exclusive OR	*EOR (immediate)* on page A8-94
		1111	0	UNPREDICTABLE	-
			1	Test Equivalence	*TEQ (immediate)* on page A8-448
1000	-	not 1111	-	Add	*ADD (immediate, Thumb)* on page A8-20
		1111	0	UNPREDICTABLE	-
			1	Compare Negative	*CMN (immediate)* on page A8-74
1010	-	-	-	Add with Carry	*ADC (immediate)* on page A8-14
1011	-	-	-	Subtract with Carry	*SBC (immediate)* on page A8-302

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A6.3.1 データ処理（修飾イミディエート）

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0		0	op				S	Rn				0				Rd											

この空間でのエンコードの割り当てを、表 A6-10 に示します。この空間に含まれる他のエンコードは未定義です。

Rn、Rd、S の「-」は、フィールドのフィールド値がデコードに影響を与えないことを示します。

これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6-10 32 ビット修飾イミディエート値データ処理命令

op	Rn	Rd	S	命令	参照先
0000	–	1111 以外	x	ビット単位論理積	*AND（イミディエート）* P. A8-34
	–	1111	0	予測不能	–
	–	1111	1	テスト	*TST（イミディエート）* P. A8-454
0001	–	–	–	ビット単位のビットクリア	*BIC（イミディエート）* P. A8-50
0010	1111 以外	–	–	ビット単位論理和	*ORR（イミディエート）* P. A8-228
	1111	–	–	移動	*MOV（イミディエート）* P. A8-194
0011	1111 以外	–	–	ビット単位否定論理和	*ORN（イミディエート）* P. A8-224
	1111	–	–	ビット単位否定	*MVN（イミディエート）* P. A8-214
0100	–	1111 以外	x	ビット単位排他的論理和	*EOR（イミディエート）* P. A8-94
		1111	0	予測不能	–
			1	等価テスト	*TEQ（イミディエート）* P. A8-448
1000	–	1111 以外	–	加算	*ADD（イミディエート、Thumb）* P. A8-20
		1111	0	予測不能	–
			1	比較否定	*CMN（イミディエート）* P. A8-74
1010	–	–	–	キャリー付き加算	*ADC（イミディエート）* P. A8-14
1011	–	–	–	キャリー付き減算	*SBC（イミディエート）* P. A8-302

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A6-10 32-bit modified immediate data-processing instructions (continued)

op	Rn	Rd	S	Instruction	See
1101	-	not 1111	-	Subtract	*SUB (immediate, Thumb)* on page A8-418
		1111	0	UNPREDICTABLE	-
			1	Compare	*CMP (immediate)* on page A8-80
1110	-	-	-	Reverse Subtract	*RSB (immediate)* on page A8-284

These instructions all have modified immediate constants, rather than a simple 12-bit binary number. This provides a more useful range of values. For details see *Modified immediate constants in Thumb instructions* on page A6-17.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A6-10 32 ビット修飾イミディエート値データ処理命令（続き）

op	Rn	Rd	S	命令	参照先
1101	–	1111 以外	–	減算	*SUB（イミディエート、Thumb）* P. A8-418
		1111	0	予測不能	–
			1	比較	*CMP（イミディエート）* P. A8-80
1110	–	–	–	逆減算	*RSB（イミディエート）* P. A8-284

これらの命令はすべて、単純な 12 ビットバイナリ数値ではなく、修飾イミディエート定数を使用します。これによって、値の範囲がより広く応用可能です。詳細については、*Thumb 命令での修飾イミディエート定数* P. A6-17 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.2 Modified immediate constants in Thumb instructions

15 14 13 12 11	10	9 8 7 6 5 4 3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
	i			imm3		a b c d e f g h

Table A6-11 shows the range of modified immediate constants available in Thumb data-processing instructions, and how they are encoded in the a, b, c, d, e, f, g, h, i, and imm3 fields in the instruction.

Table A6-11 Encoding of modified immediates in Thumb data-processing instructions

i:imm3:a	<const> [a]
0000x	00000000 00000000 00000000 abcdefgh
0001x	00000000 abcdefgh 00000000 abcdefgh [b]
0010x	abcdefgh 00000000 abcdefgh 00000000 [b]
0011x	abcdefgh abcdefgh abcdefgh abcdefgh [b]
01000	1bcdefgh 00000000 00000000 00000000
01001	01bcdefg h0000000 00000000 00000000 [c]
01010	001bcdef gh000000 00000000 00000000
01011	0001bcde fgh00000 00000000 00000000 [c]
. . .	. . 8-bit values shifted to other positions .
11101	00000000 00000000 000001bc defgh000 [c]
11110	00000000 00000000 0000001b cdefgh00
11111	00000000 00000000 00000001 bcdefgh0 [c]

a. In this table, the immediate constant value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).
b. Not available in ARM instructions. UNPREDICTABLE if abcdefgh == 00000000.
c. Not available in ARM instructions if h == 1.

Note

The range of values available in Thumb modified immediate constants is slightly different from the range of values available in ARM instructions. See *Modified immediate constants in ARM instructions* on page A5-9 for the ARM values.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A6.3.2 Thumb 命令での修飾イミディエート定数

15 14 13 12 11	10	9 8 7 6 5 4 3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
	i			imm3		a b c d e f g h

Thumb データ処理命令で使用可能な修飾イミディエート定数の範囲と、それらがこの命令内の a、b、c、d、e、f、g、h、i、imm3 の各フィールドにどのようにエンコードされるかを、表 A6-11 に示します。

表 A6-11 Thumb データ処理命令での修飾イミディエート値のエンコード

i:imm3:a	＜定数＞ [a]
0000x	00000000 00000000 00000000 abcdefgh
0001x	00000000 abcdefgh 00000000 abcdefgh [b]
0010x	abcdefgh 00000000 abcdefgh 00000000 [b]
0011x	abcdefgh abcdefgh abcdefgh abcdefgh [b]
01000	1bcdefgh 00000000 00000000 00000000
01001	01bcdefg h0000000 00000000 00000000 [c]
01010	001bcdef gh000000 00000000 00000000
01011	0001bcde fgh00000 00000000 00000000 [c]
. . .	. . 他の位置にシフトされた 8 ビット値 .
11101	00000000 00000000 000001bc defgh000 [c]
11110	00000000 00000000 0000001b cdefgh00
11111	00000000 00000000 00000001 bcdefgh0 [c]

a. この表では、abcdefgh をエンコード図に関連付けるため、イミディエート定数値をバイナリ形式で示しています。アセンブラ構文では、イミディエート値を通常の方法（デフォルトでは 10 進数）で指定します。
b. ARM 命令では使用できません。abcdefgh == 00000000 の場合、結果は予測不能です。
c. h == 1 の場合、ARM 命令では使用できません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Carry out

A logical instruction with i:imm3:a == '00xxx' does not affect the carry flag. Otherwise, a logical instruction that sets the flags sets the Carry flag to the value of bit [31] of the modified immediate constant.

Operation

```
// ThumbExpandImm()
// ================

bits(32) ThumbExpandImm(bits(12) imm12)

    // APSR.C argument to following function call does not affect the imm32 result.
    (imm32, -) = ThumbExpandImm_C(imm12, APSR.C);

    return imm32;

// ThumbExpandImm_C()
// ==================

(bits(32), bit) ThumbExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then

        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;

    else

        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

Thumb 修飾イミディエート定数に使用できる値の範囲は、ARM 命令で使用可能な値の範囲とはやや異なります。ARM で使用可能な値については、*ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

キャリーアウト

i:imm3:a == '00xxx' での論理命令は、キャリーフラグに影響を与えません。それ以外の場合、フラグをセットする論理命令によって、キャリーフラグが修飾イミディエート定数のビット [31] の値に設定されます。

動作

```
// ThumbExpandImm()
// ================

bits(32) ThumbExpandImm(bits(12) imm12)

    // APSR.C argument to following function call does not affect the imm32 result.
    (imm32, -) = ThumbExpandImm_C(imm12, APSR.C);

    return imm32;

// ThumbExpandImm_C()
// ==================

(bits(32), bit) ThumbExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then

        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = '00000000' :imm12<7:0> :'00000000' : imm12<7:0>;
            when '10'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = imm12<7:0> :'00000000' : imm12<7:0> : '00000000';
            when '11'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = imm12<7:0> :imm12<7:0> :imm12<7:0> :imm12<7:0>;
        carry_out = carry_in;

    else

        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.3 Data-processing (plain binary immediate)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0		1	op					Rn				0															

Table A6-12 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

These encodings are all available in ARMv6T2 and above.

Table A6-12 32-bit unmodified immediate data-processing instructions

op	Rn	Instruction	See
00000	not 1111	Add Wide (12-bit)	*ADD (immediate, Thumb)* on page A8-20
	1111	Form PC-relative Address	*ADR* on page A8-32
00100	-	Move Wide (16-bit)	*MOV (immediate)* on page A8-194
01010	not 1111	Subtract Wide (12-bit)	*SUB (immediate, Thumb)* on page A8-418
	1111	Form PC-relative Address	*ADR* on page A8-32
01100	-	Move Top (16-bit)	*MOVT* on page A8-200
100x0 [a]	-	Signed Saturate	*SSAT* on page A8-362
10010 [b]	-	Signed Saturate (two 16-bit)	*SSAT16* on page A8-364
10100	-	Signed Bit Field Extract	*SBFX* on page A8-308
10110	not 1111	Bit Field Insert	*BFI* on page A8-48
	1111	Bit Field Clear	*BFC* on page A8-46
110x0 [a]	-	Unsigned Saturate	*USAT* on page A8-504
11010 [b]	-	Unsigned Saturate 16	*USAT16* on page A8-506
11100	-	Unsigned Bit Field Extract	*UBFX* on page A8-466

a. In the second halfword of the instruction, bits [14:12.7:6] != 0b00000.
b. In the second halfword of the instruction, bits [14:12.7:6] == 0b00000.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.3 データ処理（普通のバイナリイミディエート）

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0		1	op					Rn				0															

この空間でのエンコードの割り当てを、表 A6-12 に示します。この空間に含まれる他のエンコードは未定義です。

これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6-12 32 ビット未修飾イミディエート値データ処理命令

op	Rn	命令	参照先
00000	1111 以外	ワイド値（12 ビット）の加算	*ADD（イミディエート、Thumb）* P. A8-20
	1111	PC 相対アドレスの生成	*ADR* P. A8-32
00100	–	ワイド値（16 ビット）の移動	*MOV（イミディエート）* P. A8-194
01010	1111 以外	ワイド値（12 ビット）の減算	*SUB（イミディエート、Thumb）* P. A8-418
	1111	PC 相対アドレスの生成	*ADR* P. A8-32
01100	–	上位ビット（16 ビット）の移動	*MOVT* P. A8-200
100x0 [a]	–	符号付き飽和	*SSAT* P. A8-362
10010 [b]	–	符号付き飽和（2 つの 16 ビット）	*SSAT16* P. A8-364
10100	–	符号付きビットフィールドの抽出	*SBFX* P. A8-308
10110	1111 以外	ビットフィールドの挿入	*BFI* P. A8-48
	1111	ビッドフィールドのクリア	*BFC* P. A8-46
110x0 [a]	–	符号なし飽和	*USAT* P. A8-504
11010 [b]	–	符号なし飽和 16	*USAT16* P. A8-506
11100	–	符号なしビットフィールドの抽出	*UBFX* P. A8-466

a. 命令の 2 番目のハーフワードでは、ビット [14:12.7:6] != 0b00000 です。
b. 命令の 2 番目のハーフワードでは、ビット [14:12.7:6] == 0b00000 です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.4 Branches and miscellaneous control

15 14 13	12 11	10 9 8 7 6 5 4	3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1	1 0	op		1	op1	op2	

Table A6-13 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A6-13 Branches and miscellaneous control instructions

op1	op	op2	Instruction	See	Variant
0x0	not x111xxx	-	Conditional branch	*B* on page A8-44	v6T2
	0111000	xx00	Move to Special Register, application level	*MSR (register)* on page A8-210	All
		xx01	Move to Special Register, system level	*MSR (register)* on page B6-14	All
		xx1x			
	0111001	-			
	0111010	-	-	*Change Processor State, and hints* on page A6-21	-
	0111011	-	-	*Miscellaneous control instructions* on page A6-21	-
	0111100	-	Branch and Exchange Jazelle	*BXJ* on page A8-64	v6T2
	0111101	-	Exception Return	*SUBS PC, LR and related instructions* on page B6-25	v6T2
	011111x	-	Move from Special Register	*MRS* on page A8-206	v6T2
000	1111111	-	Secure Monitor Call	*SMC (previously SMI)* on page B6-18	Security Extensions
010	1111111	-	Permanently UNDEFINED. This space will not be allocated in future.		
0x1	-	-	Branch	*B* on page A8-44	v6T2
1x0	-	-	Branch with Link and Exchange	*BL, BLX (immediate)* on page A8-58	v5T [a]
1x1	-	-	Branch with Link		v4T

a. UNDEFINED in ARMv4T.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A6.3.4 分岐およびその他の制御

15 14 13	12 11	10 9 8 7 6 5 4	3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1	1 0	op		1	op1	op2	

この空間でのエンコードの割り当てを、表 A6-13 に示します。この空間に含まれる他のエンコードは未定義です。

表 A6-13 分岐命令およびその他の制御命令

op1	op	op2	命令	参照先	バリアント
0x0	x111xxx 以外	–	条件付き分岐	*B* P. A8-44	v6T2
	0111000	xx00	特殊レジスタへの移動、アプリケーションレベル	*MSR（レジスタ）* P. A8-210	すべて
		xx01	特殊レジスタへの移動、システムレベル	*MSR（レジスタ）* P. B6-14	すべて
		xx1x			
	0111001	–			
	0111010	–	–	*プロセッサ状態の変更とヒント* P. A6-21	–
	0111011	–	–	*その他の制御命令* P. A6-21	–
	0111100	–	分岐と Jazelle 状態への遷移	*BXJ* P. A8-64	v6T2
	0111101	–	例外からの復帰	*SUBS PC、LR、および関連する命令* P. B6-25	v6T2
	011111x	–	特殊レジスタからの移動	*MRS* P. A8-206	v6T2
000	1111111	–	セキュア モニタコール	*SMC（以前の SMI）* P. B6-18	セキュリティ拡張機能
010	1111111	–	恒久的に未定義。この空間は将来も割り当てられません。		
0x1	–	–	分岐	*B* P. A8-44	v6T2
1x0	–	–	リンク付き分岐と状態遷移	*BL、BLX（イミディエート）* P. A8-58	v5T [a]
1x1	–	–	リンク付き分岐		v4T

a. ARMv4T では未定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Change Processor State, and hints

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0					1	0		0		op1			op2							

Table A6-14 shows the allocation of encodings in this space. Other encodings in this space are unallocated hints that execute as NOPs. These unallocated hint encodings are reserved and software must not use them.

Table A6-14 Change Processor State, and hint instructions

op1	op2	Instruction	See	Variant
not 000	-	Change Processor State	*CPS* on page B6-3	v6T2
000	00000000	No Operation hint	*NOP* on page A8-222	v6T2
	00000001	Yield hint	*YIELD* on page A8-812	v7
	00000010	Wait For Event hint	*WFE* on page A8-808	v7
	00000011	Wait For Interrupt hint	*WFI* on page A8-810	v7
	00000100	Send Event hint	*SEV* on page A8-316	v7
	1111xxxx	Debug hint	*DBG* on page A8-88	v7

Miscellaneous control instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1					1	0		0					op							

Table A6-15 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED in ARMv7. They are UNPREDICTABLE in ARMv6.

Table A6-15 Miscellaneous control instructions

op	Instruction	See	Variant
0000	Leave ThumbEE state [a]	*ENTERX, LEAVEX* on page A9-7	ThumbEE
0001	Enter ThumbEE state	*ENTERX, LEAVEX* on page A9-7	ThumbEE
0010	Clear-Exclusive	*CLREX* on page A8-70	v7

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

プロセッサ状態の変更とヒント

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0					1	0		0		op1			op2							

この空間でのエンコードの割り当てを、表 A6-14 に示します。この空間に含まれる他のエンコードは未割り当てヒントで、NOP として実行されます。これらの未割り当てヒントのエンコードは予約されているため、ソフトウェアで使用しないで下さい。

表 A6-14 プロセッサ状態変更命令およびヒント命令

op1	op2	命令	参照先	バリアント
000 以外	–	プロセッサ状態変更	*CPS* P. B6-3	v6T2
000	00000000	無操作ヒント	*NOP* P. A8-222	v6T2
	00000001	イールドヒント	*YIELD* P. A8-812	v7
	00000010	イベント待ちヒント	*WFE* P. A8-808	v7
	00000011	割り込み待ちヒント	*WFI* P. A8-810	v7
	00000100	イベント送信ヒント	*SEV* P. A8-316	v7
	1111xxxx	デバッグヒント	*DBG* P. A8-88	v7

その他の制御命令

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1					1	0		0					op							

この空間でのエンコードの割り当てを、表 A6-15 に示します。この空間に含まれる他のエンコードは、ARMv7 では未定義です。それらは ARMv6 では予測不能です。

表 A6-15 その他の制御命令

op	命令	参照先	バリアント
0000	ThumbEE 状態を終了[a]	*ENTERX*、*LEAVEX* P. A9-7	ThumbEE
0001	ThumbEE 状態を開始	*ENTERX*、*LEAVEX* P. A9-7	ThumbEE
0010	排他クリア	*CLREX* P. A8-70	v7

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A6-15 Miscellaneous control instructions (continued)

op	Instruction	See	Variant
0100	Data Synchronization Barrier	*DSB* on page A8-92	v7
0101	Data Memory Barrier	*DMB* on page A8-90	v7
0110	Instruction Synchronization Barrier	*ISB* on page A8-102	v7

a. This instruction is a NOP in Thumb state.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A6-15 その他の制御命令（続き）

op	命令	参照先	バリアント
0100	データ同期バリア	*DSB* P. A8-92	v7
0101	データ メモリバリア	*DMB* P. A8-90	v7
0110	命令同期バリア	*ISB* P. A8-102	v7

a. この命令は Thumb 状態では NOP です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.5 Load/store multiple

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	op		0		L	Rn																			

Table A6-16 shows the allocation of encodings in this space.

These encodings are all available in ARMv6T2 and above.

Table A6-16 Load/store multiple instructions

op	L	Rn	Instruction	See
00	0	-	Store Return State	*SRS* on page B6-20
	1	-	Return From Exception	*RFE* on page B6-16
01	0	-	Store Multiple (Increment After, Empty Ascending)	*STM / STMIA / STMEA* on page A8-374
	1	not 1101	Load Multiple (Increment After, Full Descending)	*LDM / LDMIA / LDMFD* on page A8-110
		1101	Pop Multiple Registers from the stack	*POP* on page A8-246
10	0	not 1101	Store Multiple (Decrement Before, Full Descending)	*STMDB / STMFD* on page A8-378
		1101	Push Multiple Registers to the stack.	*PUSH* on page A8-248
	1	-	Load Multiple (Decrement Before, Empty Ascending)	*LDMDB / LDMEA* on page A8-114
11	0	-	Store Return State	*SRS* on page B6-20
	1	-	Return From Exception	*RFE* on page B6-16

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.5 複数ロード / ストア

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	op		0		L	Rn																			

この空間でのエンコードの割り当てを、表 A6-16 に示します。

これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6-16 複数ロード / ストア命令

op	L	Rn	命令	参照先
00	0	–	復帰状態ストア	*SRS* P. B6-20
	1	–	例外からの復帰	*RFE* P. B6-16
01	0	–	複数ストア（ポストインクリメント、空上昇）	*STM*、*STMIA*、*STMEA* P. A8-374
	1	1101 以外	複数ロード（ポストインクリメント、フル下降）	*LDM/LDMIA/LDMFD* P. A8-110
		1101	スタックからの複数レジスタのポップ	*POP* P. A8-246
10	0	1101 以外	複数ストア（プリデクリメント、フル下降）	*STMDB*、*STMFD* P. A8-378
		1101	スタックへの複数レジスタのプッシュ	*PUSH* P. A8-248
	1	–	複数ロード（プリデクリメント、空上昇）	*LDMDB/LDMEA* P. A8-114
11	0	–	復帰状態ストア	*SRS* P. B6-20
	1	–	例外からの復帰	*RFE* P. B6-16

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.6 Load/store dual, load/store exclusive, table branch

15 14 13	12 11 10 9	8 7	6	5 4	3 2 1 0	15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
1 1 1	0 1 0 0	op1	1	op2	Rn		op3	

Table A6-17 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A6-17 Load/store double or exclusive, table branch

op1	op2	op3	Rn	Instruction	See	Variant
00	00	-	-	Store Register Exclusive	*STREX* on page A8-400	v6T2
	01	-	-	Load Register Exclusive	*LDREX* on page A8-142	v6T2
0x 1x	10 x0	-	-	Store Register Dual	*STRD (immediate)* on page A8-396	v6T2
0x	11	-	not 1111	Load Register Dual (immediate)	*LDRD (immediate)* on page A8-136	v6T2
1x	x1	-	not 1111			
0x	11	-	1111	Load Register Dual (literal)	*LDRD (literal)* on page A8-138	v6T2
1x	x1	-	1111			
01	00	0100	-	Store Register Exclusive Byte	*STREXB* on page A8-402	v7
		0101	-	Store Register Exclusive Halfword	*STREXH* on page A8-406	v7
		0111	-	Store Register Exclusive Doubleword	*STREXD* on page A8-404	v7
	01	0000	-	Table Branch Byte	*TBB, TBH* on page A8-446	v6T2
		0001	-	Table Branch Halfword	*TBB, TBH* on page A8-446	v6T2
		0100	-	Load Register Exclusive Byte	*LDREXB* on page A8-144	v7
		0101	-	Load Register Exclusive Halfword	*LDREXH* on page A8-148	v7
		0111	-	Load Register Exclusive Doubleword	*LDREXD* on page A8-146	v7

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.6 デュアルロード / ストア、排他ロード / ストア、テーブル分岐

15 14 13	12 11 10 9	8 7	6	5 4	3 2 1 0	15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
1 1 1	0 1 0 0	op1	1	op2	Rn		op3	

この空間でのエンコードの割り当てを、表 A6-17 に示します。この空間に含まれる他のエンコードは未定義です。

表 A6-17 ダブルまたは排他ロード / ストア、テーブル分岐

op1	op2	op3	Rn	命令	参照先	バリアント
00	00	–	–	排他レジスタストア	*STREX* P. A8-400	v6T2
	01	–	–	排他レジスタロード	*LDREX* P. A8-142	v6T2
0x 1x	10 x0	–	–	デュアル レジスタストア	*STRD（イミディエート）* P. A8-396	v6T2
0x	11	–	1111 以外	デュアル レジスタロード（イミディエート）	*LDRD（イミディエート）* P. A8-136	v6T2
1x	x1	–	1111 以外			
0x	11	–	1111	デュアル レジスタロード（リテラル）	*LDRD（リテラル）* P. A8-138	v6T2
1x	x1	–	1111			
01	00	0100	–	排他レジスタストア バイト	*STREXB* P. A8-402	v7
		0101	–	排他レジスタストア ハーフワード	*STREXH* P. A8-406	v7
		0111	–	排他レジスタストア ダブルワード	*STREXD* P. A8-404	v7
	01	0000	–	テーブル分岐バイト	*TBB*、*TBH* P. A8-446	v6T2
		0001	–	テーブル分岐ハーフワード	*TBB*、*TBH* P. A8-446	v6T2
		0100	–	排他レジスタロード バイト	*LDREXB* P. A8-144	v7
		0101	–	排他レジスタロード ハーフワード	*LDREXH* P. A8-148	v7
		0111	–	排他レジスタロード ダブルワード	*LDREXD* P. A8-146	v7

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.7 Load word

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	op1		1	0	1	Rn								op2											

Table A6-18 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

These encodings are all available in ARMv6T2 and above.

Table A6-18 Load word

op1	op2	Rn	Instruction	See
01	-	not 1111	Load Register	*LDR (immediate, Thumb)* on page A8-118
00	1xx1xx	not 1111		
	1100xx	not 1111		
	1110xx	not 1111	Load Register Unprivileged	*LDRT* on page A8-176
	000000	not 1111	Load Register	*LDR (register)* on page A8-124
0x	-	1111	Load Register	*LDR (literal)* on page A8-122

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.7 ワードロード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	op1		1	0	1	Rn								op2											

この空間でのエンコードの割り当てを、表 A6-18 に示します。この空間に含まれる他のエンコードは未定義です。

これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6-18 ワードロード

op1	op2	Rn	命令	参照先
01	–	1111 以外	レジスタロード	*LDR（イミディエート、Thumb）* P. A8-118
00	1xx1xx	1111 以外		
	1100xx	1111 以外		
	1110xx	1111 以外	非特権レジスタロード	*LDRT* P. A8-176
	000000	1111 以外	レジスタロード	*LDR（レジスタ）* P. A8-124
0x	–	1111	レジスタロード	*LDR（リテラル）* P. A8-122

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.8 Load halfword, memory hints

15 14 13	12 11 10 9	8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1	1 1 0 0	op1	0 1 1	Rn	Rt	op2		

Table A6-19 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Except where otherwise noted, these encodings are available in ARMv6T2 and above.

Table A6-19 Load halfword, preload

op1	op2	Rn	Rt	Instruction	See
0x	-	1111	not 1111	Load Register Halfword	*LDRH (literal)* on page A8-154
01	-	not 1111	not 1111	Load Register Halfword	*LDRH (immediate, Thumb)* on page A8-150
00	1xx1xx	not 1111	not 1111		
	1100xx	not 1111	not 1111		
	1110xx	not 1111	not 1111	Load Register Halfword Unprivileged	*LDRHT* on page A8-158
	000000	not 1111	not 1111	Load Register Halfword	*LDRH (register)* on page A8-156
1x	-	1111	not 1111	Load Register Signed Halfword	*LDRSH (literal)* on page A8-170
11	-	not 1111	not 1111	Load Register Signed Halfword	*LDRSH (immediate)* on page A8-168
10	1xx1xx	not 1111	not 1111		
	1100xx	not 1111	not 1111		
	1110xx	not 1111	not 1111	Load Register Signed Halfword Unprivileged	*LDRSHT* on page A8-174
	000000	not 1111	not 1111	Load Register Signed Halfword	*LDRSH (register)* on page A8-172
0x	-	1111	1111	UNPREDICTABLE	-
01	-	not 1111	1111	Preload Data with intent to Write[a]	*PLD, PLDW (immediate)* on page A8-236
00	1100xx	not 1111	1111	Preload Data with intent to Write[a]	*PLD, PLDW (immediate)* on page A8-236
	000000	not 1111	1111	Preload Data with intent to Write[a]	*PLD, PLDW (register)* on page A8-240

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A6.3.8 ハーフワードロード、メモリヒント

15 14 13	12 11 10 9	8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1	1 1 0 0	op1	0 1 1	Rn	Rt	op2		

この空間でのエンコードの割り当てを、表 A6-19 に示します。この空間に含まれる他のエンコードは未定義です。

特に記載がない限り、これらのエンコードは ARMv6T2 およびそれ以降で使用可能です。

表 A6-19 ハーフワードロード、プリロード

op1	op2	Rn	Rt	命令	参照先
0x	–	1111	1111 以外	レジスタロード ハーフワード	*LDRH（リテラル）* P. A8-154
01	–	1111 以外	1111 以外	レジスタロード ハーフワード	*LDRH（イミディエート、Thumb）* P. A8-150
00	1xx1xx	1111 以外	1111 以外		
	1100xx	1111 以外	1111 以外		
	1110xx	1111 以外	1111 以外	非特権レジスタロード ハーフワード	*LDRHT* P. A8-158
	000000	1111 以外	1111 以外	レジスタロード ハーフワード	*LDRH（レジスタ）* P. A8-156
1x	–	1111	1111 以外	レジスタロード符号付き ハーフワード	*LDRSH（リテラル）* P. A8-170
11	–	1111 以外	1111 以外	レジスタロード符号付き ハーフワード	*LDRSH（イミディエート）* P. A8-168
10	1xx1xx	1111 以外	1111 以外		
	1100xx	1111 以外	1111 以外		
	1110xx	1111 以外	1111 以外	非特権レジスタロード符号付きハーフワード	*LDRSHT* P. A8-174
	000000	1111 以外	1111 以外	レジスタロード符号付き ハーフワード	*LDRSH（レジスタ）* P. A8-172
0x	–	1111	1111	予測不能	–
01	–	1111 以外	1111	書き込みを意図したデータのプリロード[a]	*PLD、PLDW（イミディエート）* P. A8-236

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A6-19 Load halfword, preload (continued)

op1	op2	Rn	Rt	Instruction	See
00	1xx1xx	not 1111	1111	UNPREDICTABLE	-
	1110xx	not 1111	1111		
1x	-	1111	1111	Unallocated memory hint (treat as NOP)	
10	1100xx	not 1111	1111		
	000000	not 1111	1111		
10	1xx1xx	not 1111	1111	UNPREDICTABLE	-
	1110xx	not 1111	1111		
11	-	not 1111	1111	Unallocated memory hint (treat as NOP)	

a. Available in ARMv7 with the Multiprocessing Extensions. In the ARMv7 base architecture and in ARMv6T2 these are unallocated memory hints (treat as NOP).

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A6-19 ハーフワードロード、プリロード（続き）

op1	op2	Rn	Rt	命令	参照先
00	1100xx	1111 以外	1111	書き込みを意図したデータのプリロード[a]	*PLD、PLDW（イミディエート）* P. A8-236
	000000	1111 以外	1111	書き込みを意図したデータのプリロード[a]	*PLD、PLDW（レジスタ）* P. A8-240
00	1xx1xx	1111 以外	1111	予測不能	–
	1110xx	1111 以外	1111		
1x	–	1111	1111	未割り当てメモリヒント（NOP として扱う）	
10	1100xx	1111 以外	1111		
	000000	1111 以外	1111		
10	1xx1xx	1111 以外	1111	予測不能	–
	1110xx	1111 以外	1111		
11	–	1111 以外	1111	未割り当てメモリヒント（NOP として扱う）	

a. マルチプロセッシング拡張機能を持つ ARMv7 で使用可能です。ARMv7 基本アーキテクチャおよび ARMv6T2 では、未割り当てのメモリヒントです（NOP として扱われます）。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.9 Load byte, memory hints

15 14 13	12 11 10 9	8 7	6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4 3 2 1 0
1 1 1	1 1 0 0	op1	0 0	1	Rn	Rt	op2		

Table A6-20 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

These encodings are all available in ARMv6T2 and above.

Table A6-20 Load byte, preload

op1	op2	Rn	Rt	Instruction	See
0x	-	1111	not 1111	Load Register Byte	*LDRB (literal)* on page A8-130
01	-	not 1111	not 1111	Load Register Byte	*LDRB (immediate, Thumb)* on page A8-126
00	1xx1xx	not 1111	not 1111		
	1100xx	not 1111	not 1111		
	1110xx	not 1111	not 1111	Load Register Byte Unprivileged	*LDRBT* on page A8-134
	000000	not 1111	not 1111	Load Register Byte	*LDRB (register)* on page A8-132
1x	-	1111	not 1111	Load Register Signed Byte	*LDRSB (literal)* on page A8-162
11	-	not 1111	not 1111	Load Register Signed Byte	*LDRSB (immediate)* on page A8-160
10	1xx1xx	not 1111	not 1111		
	1100xx	not 1111	not 1111		
	1110xx	not 1111	not 1111	Load Register Signed Byte Unprivileged	*LDRSBT* on page A8-166
	000000	not 1111	not 1111	Load Register Signed Byte	*LDRSB (register)* on page A8-164
0x	-	1111	1111	Preload Data	*PLD (literal)* on page A8-238
01	-	not 1111	1111	Preload Data	*PLD, PLDW (immediate)* on page A8-236
00	1100xx	not 1111	1111	Preload Data	*PLD, PLDW (immediate)* on page A8-236
	000000	not 1111	1111	Preload Data	*PLD, PLDW (register)* on page A8-240
	1xx1xx	not 1111	1111	UNPREDICTABLE	-
	1110xx	not 1111	1111		

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A6.3.9 バイトロード、メモリヒント

15 14 13	12 11 10 9	8 7	6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4 3 2 1 0
1 1 1	1 1 0 0	op1	0 0	1	Rn	Rt	op2		

この空間でのエンコードの割り当てを、表 A6-20 に示します。この空間に含まれる他のエンコードは未定義です。

これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6-20 バイトロード、プリロード

op1	op2	Rn	Rt	命令	参照先
0x	–	1111	1111 以外	レジスタロード バイト	*LDRB（リテラル）* P. A8-130
01	–	1111 以外	1111 以外	レジスタロード バイト	*LDRB（イミディエート、Thumb）* P. A8-126
00	1xx1xx	1111 以外	1111 以外		
	1100xx	1111 以外	1111 以外		
	1110xx	1111 以外	1111 以外	非特権レジスタロード バイト	*LDRBT* P. A8-134
	000000	1111 以外	1111 以外	レジスタロード バイト	*LDRB（レジスタ）* P. A8-132
1x	–	1111	1111 以外	レジスタロード符号付き バイト	*LDRSB（リテラル）* P. A8-162
11	–	1111 以外	1111 以外	レジスタロード符号付き バイト	*LDRSB（イミディエート）* P. A8-160
10	1xx1xx	1111 以外	1111 以外		
	1100xx	1111 以外	1111 以外		
	1110xx	1111 以外	1111 以外	非特権レジスタロード 符号付きバイト	*LDRSBT* P. A8-166
	000000	1111 以外	1111 以外	レジスタロード符号付き バイト	*LDRSB（レジスタ）* P. A8-164
0x	–	1111	1111	データのプリロード	*PLD（リテラル）* P. A8-238
01	–	1111 以外	1111	データのプリロード	*PLD、PLDW（イミディエート）* P. A8-236

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A6-20 Load byte, preload (continued)

op1	op2	Rn	Rt	Instruction	See
1x	-	1111	1111	Preload Instruction	*PLI (immediate, literal)* on page A8-242
11	-	not 1111	1111		
10	1100xx	not 1111	1111		
	000000	not 1111	1111	Preload Instruction	*PLI (register)* on page A8-244
	1xx1xx	not 1111	1111	UNPREDICTABLE	-
	1110xx	not 1111	1111		

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A6-20 バイトロード、プリロード（続き）

op1	op2	Rn	Rt	命令	参照先
00	1100xx	1111以外	1111	データのプリロード	*PLD*、*PLDW（イミディエート）* P. A8-236
	000000	1111以外	1111	データのプリロード	*PLD*、*PLDW（レジスタ）* P. A8-240
	1xx1xx	1111以外	1111	予測不能	–
	1110xx	1111以外	1111		
1x	–	1111	1111	命令のプリロード	*PLI（イミディエート、リテラル）* P. A8-242
11	–	1111以外	1111		
10	1100xx	1111以外	1111		
	000000	1111以外	1111	命令のプリロード	*PLI（レジスタ）* P. A8-244
	1xx1xx	1111以外	1111	予測不能	–
	1110xx	1111以外	1111		

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.10 Store single data item

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	op1			0					op2															

Table A6-21 show the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

These encodings are all available in ARMv6T2 and above.

Table A6-21 Store single data item

op1	op2	Instruction	See
100	-	Store Register Byte	*STRB (immediate, Thumb)* on page A8-388
000	1xx1xx		
	1100xx		
	1110xx	Store Register Byte Unprivileged	*STRBT* on page A8-394
	0xxxxx	Store Register Byte	*STRB (register)* on page A8-392
101	-	Store Register Halfword	*STRH (immediate, Thumb)* on page A8-408
001	1xx1xx		
	1100xx		
	1110xx	Store Register Halfword Unprivileged	*STRHT* on page A8-414
001	0xxxxx	Store Register Halfword	*STRH (register)* on page A8-412
110	-	Store Register (immediate)	*STR (immediate, Thumb)* on page A8-382
010	1xx1xx		
	1100xx		
	1110xx	Store Register Unprivileged	*STRT* on page A8-416
	0xxxxx	Store Register (register)	*STR (register)* on page A8-386

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.10 単一データ項目のストア

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	op1			0					op2															

この空間でのエンコードの割り当てを、表 A6-21 に示します。この空間に含まれる他のエンコードは未定義です。

これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6-21 単一データ項目のストア

op1	op2	命令	参照先
100	–	レジスタストア バイト	*STRB（イミディエート、Thumb）* P. A8-388
000	1xx1xx		
	1100xx		
	1110xx	非特権レジスタストアバイト	*STRBT* P. A8-394
	0xxxxx	レジスタストア バイト	*STRB（レジスタ）* P. A8-392
101	–	レジスタストア ハーフワード	*STRH（イミディエート、Thumb）* P. A8-408
001	1xx1xx		
	1100xx		
	1110xx	非特権レジスタストア ハーフワード	*STRHT* P. A8-414
001	0xxxxx	レジスタストア ハーフワード	*STRH（レジスタ）* P. A8-412
110	–	レジスタストア（イミディエート）	*STR（イミディエート、Thumb）* P. A8-382
010	1xx1xx		
	1100xx		
	1110xx	非特権レジスタストア	*STRT* P. A8-416
	0xxxxx	レジスタストア（レジスタ）	*STR（レジスタ）* P. A8-386

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.11 Data-processing (shifted register)

15 14 13	12 11 10 9	8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1	0 1 0 1	op	S	Rn		Rd	

Table A6-22 shows the allocation of encodings in this space.

Other encodings in this space are UNDEFINED.

These encodings are all available in ARMv6T2 and above.

Table A6-22 Data-processing (shifted register)

op	Rn	Rd	S	Instruction	See
0000	-	not 1111	x	Bitwise AND	*AND (register)* on page A8-36
		1111	0	UNPREDICTABLE	-
			1	Test	*TST (register)* on page A8-456
0001	-	-	-	Bitwise Bit Clear	*BIC (register)* on page A8-52
0010	not 1111	-	-	Bitwise OR	*ORR (register)* on page A8-230
	1111	-	-	Move	*MOV (register)* on page A8-196
0011	not 1111	-	-	Bitwise OR NOT	*ORN (register)* on page A8-226
	1111	-	-	Bitwise NOT	*MVN (register)* on page A8-216
0100	-	not 1111	-	Bitwise Exclusive OR	*EOR (register)* on page A8-96
		1111	0	UNPREDICTABLE	-
			1	Test Equivalence	*TEQ (register)* on page A8-450
0110	-	-	-	Pack Halfword	*PKH* on page A8-234
1000	-	not 1111	-	Add	*ADD (register)* on page A8-24
		1111	0	UNPREDICTABLE	-
			1	Compare Negative	*CMN (register)* on page A8-76
1010	-	-	-	Add with Carry	*ADC (register)* on page A8-16
1011	-	-	-	Subtract with Carry	*SBC (register)* on page A8-304

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.11 データ処理（シフトしたレジスタ）

15 14 13	12 11 10 9	8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1	0 1 0 1	op	S	Rn		Rd	

この空間でのエンコードの割り当てを、表 A6-22 に示します。

この空間に含まれる他のエンコードは未定義です。

これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6-22 データ処理（シフトしたレジスタ）

op	Rn	Rd	S	命令	参照先
0000	–	1111 以外	x	ビット単位論理積	*AND（レジスタ）* P. A8-36
		1111	0	予測不能	–
			1	テスト	*TST（レジスタ）* P. A8-456
0001	–	–	–	ビット単位のビットクリア	*BIC（レジスタ）* P. A8-52
0010	1111 以外	–	–	ビット単位論理和	*ORR（レジスタ）* P. A8-230
	1111	–	–	移動	*MOV（レジスタ）* P. A8-196
0011	1111 以外	–	–	ビット単位否定論理和	*ORN（レジスタ）* P. A8-226
	1111	–	–	ビット単位否定	*MVN（レジスタ）* P. A8-216
0100	–	1111 以外	–	ビット単位排他的論理和	*EOR（レジスタ）* P. A8-96
		1111	0	予測不能	–
			1	等価テスト	*TEQ（レジスタ）* P. A8-450
0110	–	–	–	パックハーフワード	*PKH* P. A8-234
1000	–	1111 以外	–	加算	*ADD（レジスタ）* P. A8-24
		1111	0	予測不能	–
			1	比較否定	*CMN（レジスタ）* P. A8-76
1010	–	–	–	キャリー付き加算	*ADC（レジスタ）* P. A8-16
1011	–	–	–	キャリー付き減算	*SBC（レジスタ）* P. A8-304

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A6-22 Data-processing (shifted register) (continued)

op	Rn	Rd	S	Instruction	See
1101	-	not 1111	-	Subtract	*SUB (register)* on page A8-422
		1111	0	UNPREDICTABLE	-
			1	Compare	*CMP (register)* on page A8-82
1110	-	-	-	Reverse Subtract	*RSB (register)* on page A8-286

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A6-22 データ処理（シフトしたレジスタ）（続き）

op	Rn	Rd	S	命令	参照先
1101	–	1111 以外	–	減算	*SUB（レジスタ）* P. A8-422
		1111	0	予測不能	–
			1	比較	*CMP（レジスタ）* P. A8-82
1110	–	–	–	逆減算	*RSB（レジスタ）* P. A8-286

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.12 Data-processing (register)

15 14 13	12 11 10 9	8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1	1 1 0 1	0	op1	Rn	1 1 1 1		op2	

If, in the second halfword of the instruction, bits [15:12] != 0b1111, the instruction is UNDEFINED.

Table A6-23 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

These encodings are all available in ARMv6T2 and above.

Table A6-23 Data-processing (register)

op1	op2	Rn	Instruction	See
000x	0000	-	Logical Shift Left	*LSL (register)* on page A8-180
001x	0000	-	Logical Shift Right	*LSR (register)* on page A8-184
010x	0000	-	Arithmetic Shift Right	*ASR (register)* on page A8-42
011x	0000	-	Rotate Right	*ROR (register)* on page A8-280
0000	1xxx	not 1111	Signed Extend and Add Halfword	*SXTAH* on page A8-438
		1111	Signed Extend Halfword	*SXTH* on page A8-444
0001	1xxx	not 1111	Unsigned Extend and Add Halfword	*UXTAH* on page A8-518
		1111	Unsigned Extend Halfword	*UXTH* on page A8-524
0010	1xxx	not 1111	Signed Extend and Add Byte 16	*SXTAB16* on page A8-436
		1111	Signed Extend Byte 16	*SXTB16* on page A8-442
0011	1xxx	not 1111	Unsigned Extend and Add Byte 16	*UXTAB16* on page A8-516
		1111	Unsigned Extend Byte 16	*UXTB16* on page A8-522
0100	1xxx	not 1111	Signed Extend and Add Byte	*SXTAB* on page A8-434
		1111	Signed Extend Byte	*SXTB* on page A8-440
0101	1xxx	not 1111	Unsigned Extend and Add Byte	*UXTAB* on page A8-514
		1111	Unsigned Extend Byte	*UXTB* on page A8-520

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.12 データ処理（レジスタ）

15 14 13	12 11 10 9	8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1	1 1 0 1	0	op1	Rn	1 1 1 1		op2	

命令の 2 番目のハーフワードがビット [15:12] != 0b1111 の場合、その命令は未定義です。

この空間でのエンコードの割り当てを、表 A6-23 に示します。この空間に含まれる他のエンコードは未定義です。

これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6-23 データ処理（レジスタ）

op1	op2	Rn	命令	参照先
000x	0000	–	論理左シフト	*LSL（レジスタ）* P. A8-180
001x	0000	–	論理右シフト	*LSR（レジスタ）* P. A8-184
010x	0000	–	算術右シフト	*ASR（レジスタ）* P. A8-42
011x	0000	–	右ローテート	*ROR（レジスタ）* P. A8-280
0000	1xxx	1111 以外	符号付きハーフワード拡張および加算	*SXTAH* P. A8-438
		1111	符号付きハーフワード拡張	*SXTH* P. A8-444
0001	1xxx	1111 以外	符号なしハーフワード拡張および加算	*UXTAH* P. A8-518
		1111	符号なしハーフワード拡張	*UXTH* P. A8-524
0010	1xxx	1111 以外	符号付きバイト拡張および加算 16	*SXTAB16* P. A8-436
		1111	符号付きバイト拡張 16	*SXTB16* P. A8-442
0011	1xxx	1111 以外	符号なしバイト拡張および加算 16	*UXTAB16* P. A8-516
		1111	符号なしバイト拡張 16	*UXTB16* P. A8-522
0100	1xxx	1111 以外	符号付きバイト拡張および加算	*SXTAB* P. A8-434
		1111	符号付きバイト拡張	*SXTB* P. A8-440
0101	1xxx	1111 以外	符号なしバイト拡張および加算	*UXTAB* P. A8-514
		1111	符号なしバイト拡張	*UXTB* P. A8-520

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A6-23 Data-processing (register) (continued)

op1	op2	Rn	Instruction	See
1xxx	00xx	-	-	*Parallel addition and subtraction, signed* on page A6-35
	01xx	-	-	*Parallel addition and subtraction, unsigned* on page A6-36
10xx	10xx	-	-	*Miscellaneous operations* on page A6-37

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A6-23 データ処理（レジスタ）（続き）

op1	op2	Rn	命令	参照先
1xxx	00xx	–	–	*並列加算および減算、符号付き* P. A6-35
	01xx	–	–	*並列加算および減算、符号なし* P. A6-36
10xx	10xx	–	–	*その他の操作* P. A6-37

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.13 Parallel addition and subtraction, signed

15 14 13	12 11 10 9	8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1	1 1 0 1	0 1	op1		1 1 1 1		0 0	op2	

If, in the second halfword of the instruction, bits [15:12] != 0b1111, the instruction is UNDEFINED.

Table A6-24 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED. These encodings are all available in ARMv6T2 and above.

Table A6-24 Signed parallel addition and subtraction instructions

op1	op2	Instruction	See
001	00	Add 16-bit	*SADD16* on page A8-296
010	00	Add, Subtract	*SASX* on page A8-300
110	00	Subtract, Add	*SSAX* on page A8-366
101	00	Subtract 16-bit	*SSUB16* on page A8-368
000	00	Add 8-bit	*SADD8* on page A8-298
100	00	Subtract 8-bit	*SSUB8* on page A8-370
Saturating instructions			
001	01	Saturating Add 16-bit	*QADD16* on page A8-252
010	01	Saturating Add, Subtract	*QASX* on page A8-256
110	01	Saturating Subtract, Add	*QSAX* on page A8-262
101	01	Saturating Subtract 16-bit	*QSUB16* on page A8-266
000	01	Saturating Add 8-bit	*QADD8* on page A8-254
100	01	Saturating Subtract 8-bit	*QSUB8* on page A8-268
Halving instructions			
001	10	Halving Add 16-bit	*SHADD16* on page A8-318
010	10	Halving Add, Subtract	*SHASX* on page A8-322
110	10	Halving Subtract, Add	*SHSAX* on page A8-324
101	10	Halving Subtract 16-bit	*SHSUB16* on page A8-326
000	10	Halving Add 8-bit	*SHADD8* on page A8-320
100	10	Halving Subtract 8-bit	*SHSUB8* on page A8-328

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.13 並列加算および減算、符号付き

15 14 13	12 11 10 9	8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1	1 1 0 1	0 1	op1		1 1 1 1		0 0	op2	

命令の 2 番目のハーフワードがビット [15:12] != 0b1111 の場合、その命令は未定義です。

この空間でのエンコードの割り当てを、表 A6-24 に示します。この空間に含まれる他のエンコードは未定義です。これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6-24 符号付き並列加算および減算命令

op1	op2	命令	参照先
001	00	16 ビット加算	*SADD16* P. A8-296
010	00	加算、減算	*SASX* P. A8-300
110	00	減算、加算	*SSAX* P. A8-366
101	00	16 ビット減算	*SSUB16* P. A8-368
000	00	8 ビット加算	*SADD8* P. A8-298
100	00	8 ビット減算	*SSUB8* P. A8-370
飽和命令			
001	01	飽和 16 ビット加算	*QADD16* P. A8-252
010	01	飽和加算、減算	*QASX* P. A8-256
110	01	飽和減算、加算	*QSAX* P. A8-262
101	01	飽和 16 ビット減算	*QSUB16* P. A8-266
000	01	飽和 8 ビット加算	*QADD8* P. A8-254
100	01	飽和 8 ビット減算	*QSUB8* P. A8-268
半減命令			
001	10	16 ビット加算半減	*SHADD16* P. A8-318
010	10	加算、減算半減	*SHASX* P. A8-322
110	10	減算、加算半減	*SHSAX* P. A8-324
101	10	16 ビット減算半減	*SHSUB16* P. A8-326
000	10	8 ビット加算半減	*SHADD8* P. A8-320
100	10	8 ビット減算半減	*SHSUB8* P. A8-328

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.14 Parallel addition and subtraction, unsigned

15 14 13	12 11 10 9	8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1	1 1 0 1	0 1	op1		1 1 1 1		0 1	op2	

If, in the second halfword of the instruction, bits [15:12] != 0b1111, the instruction is UNDEFINED.

Table A6-25 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED. These encodings are all available in ARMv6T2 and above.

Table A6-25 Unsigned parallel addition and subtraction instructions

op1	op2	Instruction	See
001	00	Add 16-bit	*UADD16* on page A8-460
010	00	Add, Subtract	*UASX* on page A8-464
110	00	Subtract, Add	*USAX* on page A8-508
101	00	Subtract 16-bit	*USUB16* on page A8-510
000	00	Add 8-bit	*UADD8* on page A8-462
100	00	Subtract 8-bit	*USUB8* on page A8-512
Saturating instructions			
001	01	Saturating Add 16-bit	*UQADD16* on page A8-488
010	01	Saturating Add, Subtract	*UQASX* on page A8-492
110	01	Saturating Subtract, Add	*UQSAX* on page A8-494
101	01	Saturating Subtract 16-bit	*UQSUB16* on page A8-496
000	01	Saturating Add 8-bit	*UQADD8* on page A8-490
100	01	Saturating Subtract 8-bit	*UQSUB8* on page A8-498
Halving instructions			
001	10	Halving Add 16-bit	*UHADD16* on page A8-470
010	10	Halving Add, Subtract	*UHASX* on page A8-474
110	10	Halving Subtract, Add	*UHSAX* on page A8-476
101	10	Halving Subtract 16-bit	*UHSUB16* on page A8-478
000	10	Halving Add 8-bit	*UHADD8* on page A8-472
100	10	Halving Subtract 8-bit	*UHSUB8* on page A8-480

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.14 並列加算および減算、符号なし

15 14 13	12 11 10 9	8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1	1 1 0 1	0 1	op1		1 1 1 1		0 1	op2	

命令の 2 番目のハーフワードがビット [15:12] != 0b1111 の場合、その命令は未定義です。

この空間でのエンコードの割り当てを、表 A6- 25 に示します。この空間に含まれる他のエンコードは未定義です。これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6- 25 符号なし並列加算および減算命令

op1	op2	命令	参照先
001	00	16 ビット加算	*UADD16* P. A8- 460
010	00	加算、減算	*UASX* P. A8- 464
110	00	減算、加算	*USAX* P. A8- 508
101	00	16 ビット減算	*USUB16* P. A8- 510
000	00	8 ビット加算	*UADD8* P. A8- 462
100	00	8 ビット減算	*USUB8* P. A8- 512
飽和命令			
001	01	飽和 16 ビット加算	*UQADD16* P. A8- 488
010	01	飽和加算、減算	*UQASX* P. A8- 492
110	01	飽和減算、加算	*UQSAX* P. A8- 494
101	01	飽和 16 ビット減算	*UQSUB16* P. A8- 496
000	01	飽和 8 ビット加算	*UQADD8* P. A8- 490
100	01	飽和 8 ビット減算	*UQSUB8* P. A8- 498
半減命令			
001	10	16 ビット加算半減	*UHADD16* P. A8- 470
010	10	加算、減算半減	*UHASX* P. A8- 474
110	10	減算、加算半減	*UHSAX* P. A8- 476
101	10	16 ビット減算半減	*UHSUB16* P. A8- 478
000	10	8 ビット加算半減	*UHADD8* P. A8- 472
100	10	8 ビット減算半減	*UHSUB8* P. A8- 480

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.15 Miscellaneous operations

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	op1						1	1	1	1					1	0	op2					

If, in the second halfword of the instruction, bits [15:12] != 0b1111, the instruction is UNDEFINED.

Table A6-26 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED. These encodings are all available in ARMv6T2 and above.

Table A6-26 Miscellaneous operations

op1	op2	Instruction	See
00	00	Saturating Add	*QADD* on page A8-250
	01	Saturating Double and Add	*QDADD* on page A8-258
	10	Saturating Subtract	*QSUB* on page A8-264
	11	Saturating Double and Subtract	*QDSUB* on page A8-260
01	00	Byte-Reverse Word	*REV* on page A8-272
	01	Byte-Reverse Packed Halfword	*REV16* on page A8-274
	10	Reverse Bits	*RBIT* on page A8-270
	11	Byte-Reverse Signed Halfword	*REVSH* on page A8-276
10	00	Select Bytes	*SEL* on page A8-312
11	00	Count Leading Zeros	*CLZ* on page A8-72

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A6.3.15 その他の操作

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	op1						1	1	1	1					1	0	op2					

命令の 2 番目のハーフワードがビット [15:12] != 0b1111 の場合、その命令は未定義です。

この空間でのエンコードの割り当てを、表 A6-26 に示します。この空間に含まれる他のエンコードは未定義です。これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6-26 その他の操作

op1	op2	命令	参照先
00	00	飽和加算	*QADD* P. A8-250
	01	ダブル飽和加算	*QDADD* P. A8-258
	10	飽和減算	*QSUB* P. A8-264
	11	ダブル飽和減算	*QDSUB* P. A8-260
01	00	ワードのバイト反転	*REV* P. A8-272
	01	パッキングされたハーフワードのバイト反転	*REV16* P. A8-274
	10	ビット反転	*RBIT* P. A8-270
	11	符号付きハーフワードのバイト反転	*REVSH* P. A8-276
10	00	バイト選択	*SEL* P. A8-312
11	00	先行ゼロカウント	*CLZ* P. A8-72

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.16 Multiply, multiply accumulate, and absolute difference

15 14 13	12 11 10 9	8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1	1 1 0 1	1 0	op1		Ra		0 0	op2	

If, in the second halfword of the instruction, bits [7:6] != 0b00, the instruction is UNDEFINED.

Table A6-27 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED. These encodings are all available in ARMv6T2 and above.

Table A6-27 Multiply, multiply accumulate, and absolute difference operations

op1	op2	Ra	Instruction	See
000	00	not 1111	Multiply Accumulate	*MLA* on page A8-190
		1111	Multiply	*MUL* on page A8-212
	01	-	Multiply and Subtract	*MLS* on page A8-192
001	-	not 1111	Signed Multiply Accumulate (Halfwords)	*SMLABB, SMLABT, SMLATB, SMLATT* on page A8-330
		1111	Signed Multiply (Halfwords)	*SMULBB, SMULBT, SMULTB, SMULTT* on page A8-354
010	0x	not 1111	Signed Multiply Accumulate Dual	*SMLAD* on page A8-332
		1111	Signed Dual Multiply Add	*SMUAD* on page A8-352
011	0x	not 1111	Signed Multiply Accumulate (Word by halfword)	*SMLAWB, SMLAWT* on page A8-340
		1111	Signed Multiply (Word by halfword)	*SMULWB, SMULWT* on page A8-358
100	0x	not 1111	Signed Multiply Subtract Dual	*SMLSD* on page A8-342
		1111	Signed Dual Multiply Subtract	*SMUSD* on page A8-360
101	0x	not 1111	Signed Most Significant Word Multiply Accumulate	*SMMLA* on page A8-346
		1111	Signed Most Significant Word Multiply	*SMMUL* on page A8-350
110	0x	-	Signed Most Significant Word Multiply Subtract	*SMMLS* on page A8-348
111	00	not 1111	Unsigned Sum of Absolute Differences	*USAD8* on page A8-500
		1111	Unsigned Sum of Absolute Differences, Accumulate	*USADA8* on page A8-502

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.16 乗算、積和演算、および絶対値の差

15 14 13	12 11 10 9	8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1	1 1 0 1	1 0	op1		Ra		0 0	op2	

命令の 2 番目のハーフワードがビット [7:6] != 0b00 の場合、その命令は未定義です。

この空間でのエンコードの割り当てを、表 A6-27 に示します。この空間に含まれる他のエンコードは未定義です。これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6-27 乗算、積和演算、および絶対値の差の処理

op1	op2	Ra	命令	参照先
000	00	1111 以外	積和演算	*MLA* P. A8-190
		1111	乗算	*MUL* P. A8-212
	01	–	積減算	*MLS* P. A8-192
001	–	1111 以外	符号付き積和演算（ハーフワード）	*SMLABB*、*SMLABT*、*SMLATB*、*SMLATT* P. A8-330
		1111	符号付き乗算（ハーフワード）	*SMULBB*、*SMULBT*、*SMULTB*、*SMULTT* P. A8-354
010	0x	1111 以外	符号付きデュアル積和演算	*SMLAD* P. A8-332
		1111	符号付きデュアル積加算	*SMUAD* P. A8-352
011	0x	1111 以外	符号付き積和演算（ワード × ハーフワード）	*SMLAWB*、*SMLAWT* P. A8-340
		1111	符号付き乗算（ワード × ハーフワード）	*SMULWB*、*SMULWT* P. A8-358
100	0x	1111 以外	符号付きデュアル積減算	*SMLSD* P. A8-342
		1111	符号付きデュアル積減算	*SMUSD* P. A8-360
101	0x	1111 以外	符号付き最上位ワード積和演算	*SMMLA* P. A8-346
		1111	符号付き最上位ワード乗算	*SMMUL* P. A8-350
110	0x	–	符号付き最上位ワード積減算	*SMMLS* P. A8-348
111	00	1111 以外	符号なし絶対差の和	*USAD8* P. A8-500
		1111	符号なし絶対差の和、累積	*USADA8* P. A8-502

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.17 Long multiply, long multiply accumulate, and divide

15 14 13	12 11 10 9	8 7	6 5 4	3 2 1 0	15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
1 1 1	1 1 0 1	1 1	op1			op2	

Table A6-28 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A6-28 Multiply, multiply accumulate, and absolute difference operations

op1	op2	Instruction	See	Variant
000	0000	Signed Multiply Long	*SMULL* on page A8-356	v6T2
001	1111	Signed Divide	*SDIV* on page A8-310	v7-R [a]
010	0000	Unsigned Multiply Long	*UMULL* on page A8-486	v6T2
011	1111	Unsigned Divide	*UDIV* on page A8-468	v7-R [a]
100	0000	Signed Multiply Accumulate Long	*SMLAL* on page A8-334	v6T2
	10xx	Signed Multiply Accumulate Long (Halfwords)	*SMLALBB, SMLALBT, SMLALTB, SMLALTT* on page A8-336	v6T2
	110x	Signed Multiply Accumulate Long Dual	*SMLALD* on page A8-338	v6T2
101	110x	Signed Multiply Subtract Long Dual	*SMLSLD* on page A8-344	v6T2
110	0000	Unsigned Multiply Accumulate Long	*UMLAL* on page A8-484	v6T2
	0110	Unsigned Multiply Accumulate Accumulate Long	*UMAAL* on page A8-482	v6T2

a. UNDEFINED in ARMv7-A.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.17 ロング乗算、ロング積和演算、および除算

15 14 13	12 11 10 9	8 7	6 5 4	3 2 1 0	15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
1 1 1	1 1 0 1	1 1	op1			op2	

この空間でのエンコードの割り当てを、表 A6-28 に示します。この空間に含まれる他のエンコードは未定義です。

表 A6-28 乗算、積和演算、および絶対値の差の処理

op1	op2	命令	参照先	バリアント
000	0000	符号付きロング乗算	*SMULL* P. A8-356	v6T2
001	1111	符号付き除算	*SDIV* P. A8-310	v7-R [a]
010	0000	符号なしロング乗算	*UMULL* P. A8-486	v6T2
011	1111	符号なし除算	*UDIV* P. A8-468	v7-R [a]
100	0000	符号付きロング積和演算	*SMLAL* P. A8-334	v6T2
	10xx	符号付きロング積和演算（ハーフワード）	*SMLALBB*、*SMLALBT*、*SMLALTB*、*SMLALTT* P. A8-336	v6T2
	110x	符号付きデュアルロング積和演算	*SMLALD* P. A8-338	v6T2
101	110x	符号付きデュアルロング積減算	*SMLSLD* P. A8-344	v6T2
110	0000	符号なしロング積和演算	*UMLAL* P. A8-484	v6T2
	0110	符号なしロング積和和演算	*UMAAL* P. A8-482	v6T2

a. ARMv7-A では未定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.18 Coprocessor instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1		1	1	op1						Rn								coproc							op				

Table A6-29 shows the allocation of encodings in this space. These encodings are all available in ARMv6T2 and above.

Table A6-29 Coprocessor instructions

op1	op	coproc	Rn	Instructions	See
000x1x 001xxx 01xxxx	-	101x	-	Advanced SIMD, VFP	*Extension register load/store instructions* on page A7-26
000x10 001xx0 01xxx0	-	not 101x	-	Store Coprocessor	*STC, STC2* on page A8-372
000x11 001xx1 01xxx1	-	not 101x	not 1111	Load Coprocessor (immediate)	*LDC, LDC2 (immediate)* on page A8-106
000x11 001xx1 01xxx1	-	not 101x	1111	Load Coprocessor (literal)	*LDC, LDC2 (literal)* on page A8-108
00000x	-	-	-	UNDEFINED	-
00010x	-	101x	-	Advanced SIMD, VFP	*64-bit transfers between ARM core and extension registers* on page A7-32
000100	-	not 101x	-	Move to Coprocessor from two ARM core registers	*MCRR, MCRR2* on page A8-188
000101	-	not 101x	-	Move to two ARM core registers from Coprocessor	*MRRC, MRRC2* on page A8-204
10xxxx	0	101x	-	VFP	*VFP data-processing instructions* on page A7-24
		not 101x	-	Coprocessor data operations	*CDP, CDP2* on page A8-68

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A6.3.18 コプロセッサ命令

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1		1	1	op1						Rn								coproc							op				

この空間でのエンコードの割り当てを、表 A6-29 に示します。これらのエンコードはすべて、ARMv6T2 およびそれ以降で使用可能です。

表 A6-29 コプロセッサ命令

op1	op	coproc	Rn	命令	参照先
000x1x 001xxx 01xxxx	–	101x	–	アドバンスト SIMD、VFP	*拡張機能レジスタロード／ストア命令* P. A7-26
000x10 001xx0 01xxx0	–	101x 以外	–	コプロセッサストア	*STC*、*STC2* P. A8-372
000x11 001xx1 01xxx1	–	101x 以外	1111 以外	コプロセッサロード（イミディエート）	*LDC*、*LDC2（イミディエート）* P. A8-106
000x11 001xx1 01xxx1	–	101x 以外	1111	コプロセッサロード（リテラル）	*LDC*、*LDC2（リテラル）* P. A8-108
00000x	–	–	–	未定義	–
00010x	–	101x	–	アドバンスト SIMD、VFP	*ARM コアレジスタと拡張機能レジスタとの 64 ビット転送* P. A7-32
000100	–	101x 以外	–	2 つの ARM コアレジスタからコプロセッサへの移動	*MCRR*、*MCRR2* P. A8-188
000101	–	101x 以外	–	コプロセッサから 2 つの ARM コアレジスタへの移動	*MRRC*、*MRRC2* P. A8-204
10xxxx	0	101x	–	VFP	*VFP のデータ処理命令* P. A7-24
		101x 以外	–	コプロセッサのデータ操作	*CDP*、*CDP2* P. A8-68

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A6-29 Coprocessor instructions (continued)

op1	op	coproc	Rn	Instructions	See
10xxxx	1	101x	-	Advanced SIMD, VFP	*8, 16, and 32-bit transfer between ARM core and extension registers* on page A7-31
10xxx0	1	not 101x	-	Move to Coprocessor from ARM core register	*MCR, MCR2* on page A8-186
10xxx1	1	not 101x	-	Move to ARM core register from Coprocessor	*MRC, MRC2* on page A8-202
11xxxx	-	-	-	Advanced SIMD	*Advanced SIMD data-processing instructions* on page A7-10

For more information about specific coprocessors see *Coprocessor support* on page A2-68.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A6-29 コプロセッサ命令（続き）

op1	op	coproc	Rn	命令	参照先
10xxxx	1	101x	–	アドバンスト SIMD、VFP	*ARM コアレジスタと拡張機能レジスタとの 8、16、32 ビット転送* P. A7-31
10xxx0	1	101x 以外	–	ARM コアレジスタからコプロセッサへの移動	*MCR*、*MCR2* P. A8-186
10xxx1	1	101x 以外	–	コプロセッサから ARM コアレジスタへの移動	*MRC*、*MRC2* P. A8-202
11xxxx	–	–	–	アドバンスト SIMD	*アドバンスト SIMD のデータ処理命令* P. A7-10

各コプロセッサの詳細については、*コプロセッサのサポート* P. A2-68 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter A7
Advanced SIMD and VFP Instruction Encoding

This chapter gives an overview of the Advanced SIMD and VFP instruction sets. It contains the following sections:

- *Overview* on page A7-2
- *Advanced SIMD and VFP instruction syntax* on page A7-3
- *Register encoding* on page A7-8
- *Advanced SIMD data-processing instructions* on page A7-10
- *VFP data-processing instructions* on page A7-24
- *Extension register load/store instructions* on page A7-26
- *Advanced SIMD element or structure load/store instructions* on page A7-27
- *8, 16, and 32-bit transfer between ARM core and extension registers* on page A7-31
- *64-bit transfers between ARM core and extension registers* on page A7-32.

Note

- The Advanced SIMD architecture extension, its associated implementations, and supporting software, are commonly referred to as NEON™ technology.
- In the decode tables in this chapter, an entry of - for a field value means the value of the field does not affect the decoding.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 A7 章
アドバンスト SIMD および VFP 命令のエンコード

本章では、アドバンスト SIMD および VFP 命令セットの概要について説明します。本章は次のセクションから構成されています。

- *概要* P. A7-2
- *アドバンスト SIMD および VFP 命令の構文* P. A7-3
- *レジスタのエンコード* P. A7-8
- *アドバンスト SIMD のデータ処理命令* P. A7-10
- *VFP のデータ処理命令* P. A7-24
- *拡張機能レジスタロード / ストア命令* P. A7-26
- *アドバンスト SIMD のエレメントまたは構造ロード / ストア命令* P. A7-27
- *ARM コアレジスタと拡張機能レジスタとの 8、16、32 ビット転送* P. A7-31
- *ARM コアレジスタと拡張機能レジスタとの 64 ビット転送* P. A7-32

注

- アドバンスト SIMD アーキテクチャ拡張機能、それに関連する実装、対応ソフトウェアは、一般に NEON ™テクノロジと呼ばれます。
- 本章のデコード表においてフィールド値に「-」が記入されているのは、フィールドの値がデコードに影響を与えないことを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.1 Overview

All Advanced SIMD and VFP instructions are available in both ARM state and Thumb state.

A7.1.1 Advanced SIMD

The following sections describe the classes of instruction in the Advanced SIMD extension:

- *Advanced SIMD data-processing instructions* on page A7-10
- *Advanced SIMD element or structure load/store instructions* on page A7-27
- *Extension register load/store instructions* on page A7-26
- *8, 16, and 32-bit transfer between ARM core and extension registers* on page A7-31
- *64-bit transfers between ARM core and extension registers* on page A7-32.

A7.1.2 VFP

The following sections describe the classes of instruction in the VFP extension:

- *Extension register load/store instructions* on page A7-26
- *8, 16, and 32-bit transfer between ARM core and extension registers* on page A7-31
- *64-bit transfers between ARM core and extension registers* on page A7-32
- *VFP data-processing instructions* on page A7-24.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

 抄訳

A7.1 概要

アドバンスト SIMD 命令および VFP 命令はすべて、ARM 状態でも Thumb 状態でも使用可能です。

A7.1.1 アドバンスト SIMD

次に示すセクションでは、アドバンスト SIMD 拡張機能における命令のクラスについて説明します。

- *アドバンスト SIMD のデータ処理命令* P. A7-10
- *アドバンスト SIMD のエレメントまたは構造ロード / ストア命令* P. A7-27
- *拡張機能レジスタロード / ストア命令* P. A7-26
- *ARM コアレジスタと拡張機能レジスタとの 8、16、32 ビット転送* P. A7-31
- *ARM コアレジスタと拡張機能レジスタとの 64 ビット転送* P. A7-32

A7.1.2 VFP

次に示すセクションでは、VFP 拡張機能における命令のクラスについて説明します。

- *拡張機能レジスタロード / ストア命令* P. A7-26
- *ARM コアレジスタと拡張機能レジスタとの 8、16、32 ビット転送* P. A7-31
- *ARM コアレジスタと拡張機能レジスタとの 64 ビット転送* P. A7-32
- *VFP のデータ処理命令* P. A7-24

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.2 Advanced SIMD and VFP instruction syntax

Advanced SIMD and VFP instructions use the general conventions of the ARM instruction set.

Advanced SIMD and VFP data-processing instructions use the following general format:

```
V{<modifier>}<operation>{<shape>}<c><q>{.<dt>}  {<dest>,} <src1>, <src2>
```

All Advanced SIMD and VFP instructions begin with a `V`. This distinguishes Advanced SIMD vector and VFP instructions from ARM scalar instructions.

The main operation is specified in the `<operation>` field. It is usually a three letter mnemonic the same as or similar to the corresponding scalar integer instruction.

The `<c>` and `<q>` fields are standard assembler syntax fields. For details see *Standard assembler syntax fields* on page A8-7.

A7.2.1 Advanced SIMD Instruction modifiers

The `<modifier>` field provides additional variants of some instructions. Table A7-1 provides definitions of the modifiers. Modifiers are not available for every instruction.

Table A7-1 Advanced SIMD instruction modifiers

<modifier>	Meaning
Q	The operation uses saturating arithmetic.
R	The operation performs rounding.
D	The operation doubles the result (before accumulation, if any).
H	The operation halves the result.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.2 アドバンスト SIMD および VFP 命令の構文

アドバンスト SIMD 命令と VFP 命令は、ARM 命令セットの一般的な規則を使用します。

アドバンスト SIMD と VFP のデータ処理命令は、次の一般的なフォーマットを使用します。

```
V{<modifier>}<operation>{<shape>}<c><q>{.<dt>}  {<dest>,} <src1>, <src2>
```

アドバンスト SIMD 命令と VFP 命令はすべて v で始まります。これによって、アドバンスト SIMD ベクタ命令および VFP 命令を ARM スカラ命令と区別することができます。

主な動作は `<operation>` フィールドに指定されています。通常、これは 3 文字のニーモニックで、対応するスカラ整数命令と同じか類似しています。

`<c>` および `<q>` フィールドは、標準アセンブラ構文のフィールドです。詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

A7.2.1 アドバンスト SIMD 命令の修飾子

`<modifier>` フィールドは、一部の命令のバリアントを作ります。修飾子の定義を、表 A7-1 に示します。修飾子はすべての命令に使用できるわけではありません。

表 A7-1 アドバンスト SIMD 命令の修飾子

< 修飾子 >	意味
Q	この操作では、飽和算術演算を使用します。
R	この操作は丸めを実行します。
D	この操作は、結果を 2 倍にします（累積を行う場合はその前に）。
H	この操作は、結果を半分にします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.2.2 Advanced SIMD Operand shapes

The <shape> field provides additional variants of some instructions. Table A7-2 provides definitions of the shapes. Operand shapes are not available for every instruction.

Table A7-2 Advanced SIMD operand shapes

<shape>	Meaning	Typical register shape	
(none)	The operands and result are all the same width.	Dd, Dn, Dm	Qd, Qn, Qm
L	Long operation - result is twice the width of both operands	Qd, Dn, Dm	
N	Narrow operation - result is half the width of both operands	Dd, Qn, Qm	
W	Wide operation - result and first operand are twice the width of the second operand	Qd, Qn, Dm	

A7.2.3 Data type specifiers

The <dt> field normally contains one data type specifier. This indicates the data type contained in

- the second operand, if any
- the operand, if there is no second operand
- the result, if there are no operand registers.

The data types of the other operand and result are implied by the <dt> field combined with the instruction shape. For information about data type formats see *Data types supported by the Advanced SIMD extension* on page A2-25.

In the instruction syntax descriptions in Chapter A8 *Instruction Details*, the <dt> field is usually specified as a single field. However, where more convenient, it is sometimes specified as a concatenation of two fields, <type><size>.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.2.2 アドバンスト SIMD オペランドの形状

一部の命令では、<shape> フィールドによって追加のバリアントが提供されます。形状の定義を、表 A7-2 に示します。オペランドの形状はすべての命令に使用できるわけではありません。

表 A7-2 アドバンスト SIMD オペランドの形状

<shape>	意味	標準的なレジスタの形状	
（なし）	オペランドと結果はすべて同じ幅です。	Dd、Dn、Dm	Qd、Qn、Qm
L	ロング操作－結果は両方のオペランドの 2 倍の幅になります。	Qd、Dn、Dm	
N	縮小操作－結果は両方のオペランドの 2 倍の幅になります。	Dd、Qn、Qm	
W	ワイド操作－結果と第 1 オペランドが、第 2 オペランドの 2 倍の幅になります。	Qd、Qn、Dm	

A7.2.3 データ型指定子

<dt> フィールドは通常、1 つのデータ型指定子を含みます。この指定子は、次のいずれかのデータ型を示します。

- 第 2 オペランド（存在する場合）
- 第 2 オペランドがない場合、オペランド
- オペランドレジスタがない場合、結果

他のオペランドや結果のデータ型は、<dt> フィールドと命令形状の組み合わせによって暗黙的に示されます。データ型フォーマットの詳細については、*アドバンスト SIMD 拡張機能でサポートされるデータ型* P. A2-25 を参照して下さい。

第 A8 章 *「命令の詳細」*の命令構文の説明において、<dt> フィールドは通常 1 つのフィールドとして指定されます。しかし、便宜上、連結した 2 つのフィールドの <type><size> として指定される場合もあります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Syntax flexibility

There is some flexibility in the data type specifier syntax:

- You can specify three data types, specifying the result and both operand data types. For example:

  ```
  VSUBW.I16.I16.S8 Q3,Q5,D0
  ```

  instead of:

  ```
  VSUBW.S8 Q3,Q5,D0
  ```

- You can specify two data types, specifying the data types of the two operands. The data type of the result is implied by the instruction shape.
- You can specify two data types, specifying the data types of the single operand and the result.
- Where an instruction requires a less specific data type, you can instead specify a more specific type, as shown in Table A7-3.
- Where an instruction does not require a data type, you can provide one.
- The F32 data type can be abbreviated to F.
- The F64 data type can be abbreviated to D.

In all cases, if you provide additional information, the additional information must match the instruction shape. Disassembly does not regenerate this additional information.

Table A7-3 Data type specification flexibility

Specified data type	Permitted more specific data types				
None	Any				
.I<size>	-	.S<size>	.U<size>	-	-
.8	.I8	.S8	.U8	.P8	-
.16	.I16	.S16	.U16	.P16	.F16
.32	.I32	.S32	.U32	-	.F32 or .F
.64	.I64	.S64	.U64	-	.F64 or .D

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

構文の柔軟性

データ型指定子の構文には、多少の柔軟性があります。

- 3 つのデータ型を指定し、結果と両オペランドのデータ型を指定することができます。次に例を示します。

  ```
  VSUBW.I16.I16.S8 Q3,Q5,D0
  ```

  は、次の形式の代わりに使用できます。

  ```
  VSUBW.S8 Q3,Q5,D0
  ```

- 2 つのデータ型を指定し、2 つのオペランドのデータ型を指定できます。結果のデータ型は、命令の形状によって暗黙的に示されます。
- 2 つのデータ型を指定し、1 つのオペランドと結果のデータ型を指定できます。
- 命令がそれほど具体的なデータ型を要求しない場合、表 A7-3 に示すように、より具体的な型を指定することができます。
- 命令が特定のデータ型を要求しない場合、データ型を指定することができます。
- F32 データ型は F と省略できます。
- F64 データ型は D と省略できます。

どのような場合でも、追加情報を提供する際は、その追加情報が命令の形状に一致している必要があります。逆アセンブリではこの追加情報が再生成されません。

表 A7-3 データ型指定の柔軟性

指定されたデータ型	許容される具体的なデータ型				
なし	任意				
.I<size>	–	.S<size>	.U<size>	–	–
.8	.I8	.S8	.U8	.P8	–
.16	.I16	.S16	.U16	.P16	.F16
.32	.I32	.S32	.U32	–	.F32 または .F
.64	.I64	.S64	.U64	–	.F64 または .D

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.2.4 Register specifiers

The <dest>, <src1>, and <src2> fields contain register specifiers, or in some cases scalar specifiers or register lists. Table A7-4 shows the register and scalar specifier formats that appear in the instruction descriptions.

If <dest> is omitted, it is the same as <src1>.

Table A7-4 Advanced SIMD and VFP register specifier formats

<specifier>	Usual meaning [a]
<Qd>	A quadword destination register for the result vector (Advanced SIMD only).
<Qn>	A quadword source register for the first operand vector (Advanced SIMD only).
<Qm>	A quadword source register for the second operand vector (Advanced SIMD only).
<Dd>	A doubleword destination register for the result vector.
<Dn>	A doubleword source register for the first operand vector.
<Dm>	A doubleword source register for the second operand vector.
<Sd>	A singleword destination register for the result vector (VFP only).
<Sn>	A singleword source register for the first operand vector (VFP only).
<Sm>	A singleword source register for the second operand vector (VFP only).
<Dd[x]>	A destination scalar for the result. Element x of vector <Dd>. (Advanced SIMD only).
<Dn[x]>	A source scalar for the first operand. Element x of vector <Dn>. (Advanced SIMD only).
<Dm[x]>	A source scalar for the second operand. Element x of vector <Dm>. (Advanced SIMD only).
<Rd>	An ARM core register. Can be source or destination.
<Rm>	An ARM core register. Can be source or destination.

a. In some instructions the roles of registers are different.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.2.4 レジスタ指定子

<dest>、<src1>、<src2> フィールドは、レジスタ指定子、あるいは場合によってはスカラ指定子やレジスタリストを含みます。命令の説明に使用されるレジスタおよびスカラ指定子のフォーマットを、表 A7-4 に示します。

<dest> が省略されている場合は、<src1> と同じです。

表 A7-4 アドバンスト SIMD および VFP レジスタ指定子のフォーマット

＜指定子＞	通常の意味[a]
<Qd>	結果ベクタに使用するクワッドワードのデスティネーションレジスタ（アドバンスト SIMD のみ）
<Qn>	第 1 オペランドのベクタに使用するクワッドワードのソースレジスタ（アドバンスト SIMD のみ）
<Qm>	第 2 オペランドのベクタに使用するクワッドワードのソースレジスタ（アドバンスト SIMD のみ）
<Dd>	結果ベクタに使用するダブルワードのデスティネーションレジスタ
<Dn>	第 1 オペランドのベクタに使用するダブルワードのソースレジスタ
<Dm>	第 2 オペランドのベクタに使用するダブルワードのソースレジスタ
<Sd>	結果ベクタに使用するシングルワードのデスティネーションレジスタ（VFP のみ）
<Sn>	第 1 オペランドのベクタに使用するシングルワードのソースレジスタ（VFP のみ）
<Sm>	第 2 オペランドのベクタに使用するシングルワードのソースレジスタ（VFP のみ）
<Dd[x]>	結果に使用するデスティネーションスカラ。ベクタ <Dd> のエレメント x です（アドバンスト SIMD のみ）。
<Dn[x]>	第 1 オペランドに使用するソーススカラ。ベクタ <Dn> のエレメント x です（アドバンスト SIMD のみ）。
<Dm[x]>	第 2 オペランドに使用するソーススカラ。ベクタ <Dm> のエレメント x です（アドバンスト SIMD のみ）。
<Rd>	ARM コアレジスタ。ソースの場合もデスティネーションの場合もあります。
<Rm>	ARM コアレジスタ。ソースの場合もデスティネーションの場合もあります。

a. 一部の命令では、レジスタの役割が異なります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.2.5 Register lists

A register list is a list of register specifiers separated by commas and enclosed in brackets { and }. There are restrictions on what registers can appear in a register list. These restrictions are described in the individual instruction descriptions. Table A7-5 shows some register list formats, with examples of actual register lists corresponding to those formats.

Note

Register lists must not wrap around the end of the register bank.

Syntax flexibility

There is some flexibility in the register list syntax:

- Where a register list contains consecutive registers, they can be specified as a range, instead of listing every register, for example `{D0-D3}` instead of `{D0,D1,D2,D3}`.
- Where a register list contains an even number of consecutive doubleword registers starting with an even numbered register, it can be written as a list of quadword registers instead, for example `{Q1,Q2}` instead of `{D2-D5}`.
- Where a register list contains only one register, the enclosing braces can be omitted, for example `VLD1.8 D0,[R0]` instead of `VLD1.8 {D0},[R0]`.

Table A7-5 Example register lists

Format	Example	Alternative
`{<Dd>}`	`{D3}`	`D3`
`{<Dd>,<Dd+1>,<Dd+2>}`	`{D3,D4,D5}`	`{D3-D5}`
`{<Dd[x]>,<Dd+2[x]}`	`{D0[3],D2[3]}`	-
`{<Dd[]>}`	`{D7[]}`	`D7[]`

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.2.5 レジスタリスト

レジスタリストは、カンマで区切られ、{ と } で囲まれたレジスタ指定子のリストです。レジスタリストに含めるレジスタには制限があります。この制限は、各命令の説明に記載されています。いくつかのレジスタリスト フォーマットと、そのフォーマットに対応する実際のレジスタリストの例を、表 A7-5 に示します。

注

レジスタリストはレジスタバンクの終わりをラップアラウンドできません。

構文の柔軟性

レジスタリストの構文には、多少の柔軟性があります。

- レジスタリストが連続するレジスタを含む場合、すべてのレジスタを列記するのではなく、レジスタを範囲として指定することができます。例えば、`{D0,D1,D2,D3}` は `{D0-D3}` のように記載できます。
- レジスタリストが偶数番号のレジスタから始まる偶数個の連続するダブルワードレジスタを含む場合は、クワッドワードのレジスタのリストとして記述できます。例えば、`{D2-D5}` は `{Q1,Q2}` のように記載できます。
- レジスタリストにレジスタが 1 つだけ含まれる場合、括弧は省略できます。例えば、`VLD1.8 {D0},[R0]` は `VLD1.8 D0,[R0]` のように記載できます。

表 A7-5 レジスタリストの例

フォーマット	例	別の記載方法
`{<Dd>}`	`{D3}`	`D3`
`{<Dd>,<Dd+1>,<Dd+2>}`	`{D3,D4,D5}`	`{D3-D5}`
`{<Dd[x]>,<Dd+2[x]}`	`{D0[3],D2[3]}`	–
`{<Dd[]>}`	`{D7[]}`	`D7[]`

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.3 Register encoding

Advanced SIMD registers are either *quadword* (128 bits wide) or *doubleword* (64 bits wide). Some instructions have options for either doubleword or quadword registers. This is normally encoded in Q (bit [6]) as Q = 0 for doubleword operations, Q = 1 for quadword operations.

VFP registers are either double-precision (64 bits wide) or single-precision (32 bits wide). This is encoded in the sz field (bit [8]) as sz = 1 for double-precision operations, or sz = 0 for single-precision operations.

Some instructions use only one or two registers, and use the unused register fields as additional opcode bits.

Table A7-6 shows the encodings for the registers.

Thumb encoding

15 14 13 12 11 10 9 8 7	6	5 4	3 2 1 0	15 14 13 12 11 10 9	8	7	6	5	4	3 2 1 0
	D		Vn	Vd	sz	N	Q	M		Vm

ARM encoding

31 30 29 28 27 26 25 24 23	22	21 20	19 18 17 16	15 14 13 12 11 10 9	8	7	6	5	4	3 2 1 0
	D		Vn	Vd	sz	N	Q	M		Vm

Table A7-6 Encoding of register numbers

Register mnemonic	Usual usage	Register number encoded in	Notes [a]	Used in
<Qd>	Destination (quadword)	D, Vd (bits [22,15:13])	bit [12] == 0	Adv. SIMD
<Qn>	First operand (quadword)	N, Vn (bits [7,19:17])	bit [16] == 0	Adv. SIMD
<Qm>	Second operand (quadword)	M, Vm (bits [5,3:1])	bit [0] == 0	Adv. SIMD
<Dd>	Destination (doubleword)	D, Vd (bits [22,15:12])	-	Both
<Dn>	First operand (doubleword)	N, Vn (bits [7,19:16])	-	Both
<Dm>	Second operand (doubleword)	M, Vm (bits [5,3:0])	-	Both
<Sd>	Destination (single-precision)	Vd, D (bits [15:12,22])	-	VFP
<Sn>	First operand (single-precision)	Vn, N (bits [19:16,7])	-	VFP
<Sm>	Second operand (single-precision)	Vm, M (bits [3:0,5])	-	VFP

a. If one of these bits is 1, the instruction is UNDEFINED.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.3 レジスタのエンコード

アドバンスト SIMD レジスタは、*クワッドワード*（128 ビット幅）または*ダブルワード*（64 ビット幅）のいずれかです。一部の命令では、ダブルワードレジスタまたはクワッドワードレジスタのいずれかを選択できます。これは通常、Q（ビット [6]）にエンコードされます。Q = 0 はダブルワード演算、Q = 1 はクワッドワード演算です。

VFP レジスタは、倍精度（64 ビット幅）または単精度（32 ビット幅）のいずれかです。これは、sz フィールド（ビット [8]）にエンコードされます。sz = 1 は倍精度演算、sz = 0 は単精度演算です。

1 つまたは 2 つのレジスタしか使用せず、未使用のレジスタフィールドを追加のオペコードビットとして使用する命令もあります。

レジスタのエンコードを、表 A7-6 に示します。

Thumb エンコード

15 14 13 12 11 10 9 8 7	6	5 4	3 2 1 0	15 14 13 12 11 10 9	8	7	6	5	4	3 2 1 0
	D		Vn	Vd	sz	N	Q	M		Vm

ARM エンコード

31 30 29 28 27 26 25 24 23	22	21 20	19 18 17 16	15 14 13 12 11 10 9	8	7	6	5	4	3 2 1 0
	D		Vn	Vd	sz	N	Q	M		Vm

表 A7-6 レジスタ番号のエンコード

レジスタニーモニック	通常の用法	レジスタ番号のエンコードされる場所	注[a]	使用される命令
<Qd>	デスティネーション（クワッドワード）	D、Vd（ビット [22、15:13]）	ビット [12]＝0	アドバンスト SIMD
<Qn>	第 1 オペランド（クワッドワード）	N、Vn（ビット [7、19:17]）	ビット [16]＝0	アドバンスト SIMD
<Qm>	第 2 オペランド（クワッドワード）	M、Vm（ビット [5、3:1]）	ビット [0]＝0	アドバンスト SIMD
<Dd>	デスティネーション（ダブルワード）	D、Vd（ビット [22、15:12]）	－	両方
<Dn>	第 1 オペランド（ダブルワード）	N、Vn（ビット [7、19:16]）	－	両方
<Dm>	第 2 オペランド（ダブルワード）	M、Vm（ビット [5、3:0]）	－	両方
<Sd>	デスティネーション（単精度）	Vd、D（ビット [15:12、22]）	－	VFP
<Sn>	第 1 オペランド（単精度）	Vn、N（ビット [19:16、7]）	－	VFP
<Sm>	第 2 オペランド（単精度）	Vm、M（ビット [3:0、5]）	－	VFP

a. ビットの 1 つが 1 のとき、命令は未定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.3.1 Advanced SIMD scalars

Advanced SIMD scalars can be 8-bit, 16-bit, 32-bit, or 64-bit. Instructions other than multiply instructions can access any element in the register set. The instruction syntax refers to the scalars using an index into a doubleword vector. The descriptions of the individual instructions contain details of the encodings.

Table A7-7 shows the form of encoding for scalars used in multiply instructions. These instructions cannot access scalars in some registers. The descriptions of the individual instructions contain cross references to this section where appropriate.

32-bit Advanced SIMD scalars, when used as single-precision floating-point numbers, are equivalent to VFP single-precision registers. That is, `Dm[x]` in a 32-bit context (0 <= `m` <= 15, 0 <= `x` <=1) is equivalent to `S[2m + x]`.

Table A7-7 Encoding of scalars in multiply instructions

Scalar mnemonic	Usual usage	Scalar size	Register specifier	Index specifier	Accessible registers
`<Dm[x]>`	Second operand	16-bit	Vm[2:0]	M, Vm[3]	D0-D7
		32-bit	Vm[3:0]	M	D0-D15

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.3.1 アドバンスト SIMD スカラ

アドバンスト SIMD スカラは、8 ビット、16 ビット、32 ビット、64 ビットの場合があります。乗算命令以外の命令は、レジスタセットのどのエレメントにもアクセスできます。命令構文では、ダブルワードベクタへのインデクスを使用してスカラを参照します。各命令の説明には、エンコードの詳細が含まれます。

乗算命令に使用されるスカラのエンコード形式を、表 A7- 7 に示します。これらの命令は、一部のレジスタのスカラにアクセスできません。各命令の説明には、適宜、このセクションへの相互参照が含まれます。

32 ビットのアドバンスト SIMD スカラは、単精度の浮動小数点数として使用された場合、VFP の単精度レジスタに等しくなります。つまり、`Dm[x]` が 32 ビットのコンテキスト (0 <= `m` <= 15、0 <= `x` <=1) にある場合は、`S[2m + x]` と等価です。

表 A7- 7 乗算命令におけるスカラのエンコード

スカラニーモニック	通常の用法	スカラサイズ	レジスタ指定子	インデクス指定子	アクセス可能なレジスタ
`<Dm[x]>`	第 2 オペランド	16 ビット	Vm[2:0]	M、Vm[3]	D0 ～ D7
		32 ビット	Vm[3:0]	M	D0 ～ D15

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4 Advanced SIMD data-processing instructions

Thumb encoding

15 14 13	12	11 10 9 8	7 6 5 4 3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1	U	1 1 1 1	A		B	C	

ARM encoding

31 30 29 28	27 26 25	24	23 22 21 20 19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	0 0 1	U	A		B	C	

Table A7-8 shows the encoding for Advanced SIMD data-processing instructions. Other encodings in this space are UNDEFINED.

In these instructions, the U bit is in a different location in ARM and Thumb instructions. This is bit [12] of the first halfword in the Thumb encoding, and bit [24] in the ARM encoding. Other variable bits are in identical locations in the two encodings, after adjusting for the fact that the ARM encoding is held in memory as a single word and the Thumb encoding is held as two consecutive halfwords.

The ARM instructions can only be executed unconditionally. The Thumb instructions can be executed conditionally by using the IT instruction. For details see *IT* on page A8-104.

Table A7-8 Data-processing instructions

U	A	B	C	See
-	0xxxx	-	-	*Three registers of the same length* on page A7-12
	1x000	-	0xx1	*One register and a modified immediate value* on page A7-21
	1x001	-	0xx1	*Two registers and a shift amount* on page A7-17
	1x01x	-	0xx1	
	1x1xx	-	0xx1	
	1xxxx	-	1xx1	
	1x0xx	-	x0x0	*Three registers of different lengths* on page A7-15
	1x10x	-	x0x0	
	1x0xx	-	x1x0	*Two registers and a scalar* on page A7-16
	1x10x	-	x1x0	

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4 アドバンスト SIMD のデータ処理命令

Thumb エンコード

15 14 13	12	11 10 9 8	7 6 5 4 3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1	U	1 1 1 1	A		B	C	

ARM エンコード

31 30 29 28	27 26 25	24	23 22 21 20 19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	0 0 1	U	A		B	C	

アドバンスト SIMD データ処理命令のエンコードを、表 A7-8 に示します。この空間に含まれる他のエンコードは未定義です。

これらの命令において、U ビットは ARM 命令と Thumb 命令で異なる位置にあります。すなわち、Thumb エンコードでは最初のハーフワードのビット [12]、ARM エンコードではビット [24] です。他の可変ビットは 2 つのエンコードで同じ位置にあります。ただし、ARM エンコードは単一ワードとしてメモリに保持され、Thumb エンコードは 2 つの連続するハーフワードとして保持される点が異なります。

ARM 命令は無条件でしか実行できません。Thumb 命令は IT 命令を使用して条件付き実行が可能です。詳細については、*IT* P. A8-104 を参照して下さい。

表 A7-8 データ処理命令

U	A	B	C	参照先
–	0xxxx	–	–	*同じ長さの 3 つのレジスタ* P. A7-12
	1x000	–	0xx1	*1 つのレジスタと修飾イミディエート値* P. A7-21
	1x001	–	0xx1	*2 つのレジスタとシフト量* P. A7-17
	1x01x	–	0xx1	
	1x1xx	–	0xx1	
	1xxxx	–	1xx1	
	1x0xx	–	x0x0	*異なる長さの 3 つのレジスタ* P. A7-15
	1x10x	–	x0x0	
	1x0xx	–	x1x0	*2 つのレジスタと 1 つのスカラ* P. A7-16
	1x10x	–	x1x0	

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A7-8 Data-processing instructions (continued)

U	A	B	C	See
0	1x11x	-	xxx0	Vector Extract, *VEXT* on page A8-598
1	1x11x	0xxx	xxx0	*Two registers, miscellaneous* on page A7-19
		10xx	xxx0	Vector Table Lookup, *VTBL*, *VTBX* on page A8-798
		1100	0xx0	Vector Duplicate, *VDUP (scalar)* on page A8-592

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A7-8 データ処理命令（続き）

U	A	B	C	参照先
0	1x11x	–	xxx0	ベクタ抽出、*VEXT* P. A8-598
1	1x11x	0xxx	xxx0	*2 つのレジスタ、その他* P. A7-19
		10xx	xxx0	ベクタテーブル ルックアップ、*VTBL*、*VTBX* P. A8-798
		1100	0xx0	ベクタ複製、*VDUP（スカラ）* P. A8-592

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4.1 Three registers of the same length

Thumb encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0		C										A							B				

ARM encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0		C										A							B				

Table A7-9 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A7-9 Three registers of the same length

A	B	U	C	Instruction	See
0000	0	-	-	Vector Halving Add	*VHADD, VHSUB* on page A8-600
	1	-	-	Vector Saturating Add	*VQADD* on page A8-700
0001	0	-	-	Vector Rounding Halving Add	*VRHADD* on page A8-734
	1	0	00	Vector Bitwise AND	*VAND (register)* on page A8-544
			01	Vector Bitwise Bit Clear (AND complement)	*VBIC (register)* on page A8-548
			10	Vector Bitwise OR (if source registers differ)	*VORR (register)* on page A8-680
				Vector Move (if source registers identical)	*VMOV (register)* on page A8-642
			11	Vector Bitwise OR NOT	*VORN (register)* on page A8-676
		1	00	Vector Bitwise Exclusive OR	*VEOR* on page A8-596
			01	Vector Bitwise Select	*VBIF, VBIT, VBSL* on page A8-550
			10	Vector Bitwise Insert if True	*VBIF, VBIT, VBSL* on page A8-550
			11	Vector Bitwise Insert if False	*VBIF, VBIT, VBSL* on page A8-550
0010	0	-	-	Vector Halving Subtract	*VHADD, VHSUB* on page A8-600
	1	-	-	Vector Saturating Subtract	*VQSUB* on page A8-724
0011	0	-	-	Vector Compare Greater Than	*VCGT (register)* on page A8-560
	1	-	-	Vector Compare Greater Than or Equal	*VCGE (register)* on page A8-556

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4.1 同じ長さの 3 つのレジスタ

Thumb エンコード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0		C										A							B				

ARM エンコード

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0		C										A							B				

この空間でのエンコードの割り当てを、表 A7-9 に示します。この空間に含まれる他のエンコードは未定義です。

表 A7-9 同じ長さの 3 つのレジスタ

A	B	U	C	命令	参照先
0000	0	–	–	ベクタ加算半減	*VHADD*、*VHSUB* P. A8-600
	1	–	–	ベクタ飽和加算	*VQADD* P. A8-700
0001	0	–	–	ベクタ丸め加算半減	*VRHADD* P. A8-734
	1	0	00	ベクタビット単位論理積	*VAND（レジスタ）* P. A8-544
			01	ベクタビット単位ビットクリア（補数論理積）	*VBIC（レジスタ）* P. A8-548
			10	ベクタビット単位論理和（ソースレジスタが異なる場合）	*VORR（レジスタ）* P. A8-680
				ベクタ移動（ソースレジスタが同一の場合）	*VMOV（レジスタ）* P. A8-642
			11	ベクタビット単位否定論理和	*VORN（レジスタ）* P. A8-676
		1	00	ベクタビット単位排他的論理和	*VEOR* P. A8-596
			01	ベクタビット単位選択	*VBIF*、*VBIT*、*VBSL* P. A8-550
			10	TRUE 時ベクタビット単位挿入	*VBIF*、*VBIT*、*VBSL* P. A8-550
			11	FALSE 時ベクタビット単位挿入	*VBIF*、*VBIT*、*VBSL* P. A8-550

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A7-9 Three registers of the same length (continued)

A	B	U	C	Instruction	See
0100	0	-	-	Vector Shift Left	*VSHL (register)* on page A8-752
	1	-	-	Vector Saturating Shift Left	*VQSHL (register)* on page A8-718
0101	0	-	-	Vector Rounding Shift Left	*VRSHL* on page A8-736
	1	-	-	Vector Saturating Rounding Shift Left	*VQRSHL* on page A8-714
0110	-	-	-	Vector Maximum or Minimum	*VMAX, VMIN (integer)* on page A8-630
0111	0	-	-	Vector Absolute Difference	*VABD, VABDL (integer)* on page A8-528
	1	-	-	Vector Absolute Difference and Accumulate	*VABA, VABAL* on page A8-526
1000	0	0	-	Vector Add	*VADD (integer)* on page A8-536
		1	-	Vector Subtract	*VSUB (integer)* on page A8-788
	1	0	-	Vector Test Bits	*VTST* on page A8-802
		1	-	Vector Compare Equal	*VCEQ (register)* on page A8-552
1001	0	-	-	Vector Multiply Accumulate or Subtract	*VMLA, VMLAL, VMLS, VMLSL (integer)* on page A8-634
	1	-	-	Vector Multiply	*VMUL, VMULL (integer and polynomial)* on page A8-662
1010	-	-	-	Vector Pairwise Maximum or Minimum	*VPMAX, VPMIN (integer)* on page A8-690
1011	0	0	-	Vector Saturating Doubling Multiply Returning High Half	*VQDMULH* on page A8-704
		1	-	Vector Saturating Rounding Doubling Multiply Returning High Half	*VQRDMULH* on page A8-712
	1	0	-	Vector Pairwise Add	*VPADD (integer)* on page A8-684

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A7-9 同じ長さの 3 つのレジスタ（続き）

A	B	U	C	命令	参照先
0010	0	–	–	ベクタ減算半減	*VHADD*、*VHSUB* P. A8-600
	1	–	–	ベクタ飽和減算	*VQSUB* P. A8-724
0011	0	–	–	ベクタ比較より大きい	*VCGT（レジスタ）* P. A8-560
	1	–	–	ベクタ比較以上	*VCGE（レジスタ）* P. A8-556
0100	0	–	–	ベクタ左シフト	*VSHL（レジスタ）* P. A8-752
	1	–	–	ベクタ飽和左シフト	*VQSHL（レジスタ）* P. A8-718
0101	0	–	–	ベクタ丸め付き左シフト	*VRSHL* P. A8-736
	1	–	–	ベクタ飽和丸め付き左シフト	*VQRSHL* P. A8-714
0110	–	–	–	ベクタ最大または最小	*VMAX*、*VMIN（整数）* P. A8-630
0111	0	–	–	ベクタ絶対値の差	*VABD*、*VABDL（整数）* P. A8-528
	1	–	–	ベクタ絶対値の差および累積	*VABA*、*VABAL* P. A8-526
1000	0	0	–	ベクタ加算	*VADD（整数）* P. A8-536
		1	–	ベクタ減算	*VSUB（整数）* P. A8-788
	1	0	–	ベクタ テストビット	*VTST* P. A8-802
		1	–	ベクタ比較等価	*VCEQ（レジスタ）* P. A8-552
1001	0	–	–	ベクタ積和または積減算	*VMLA*、*VMLAL*、*VMLS*、*VMLSL（整数）* P. A8-634
	1	–	–	ベクタ乗算	*VMUL*、*VMULL（整数および多項式）* P. A8-662
1010	–	–	–	ベクタペア単位最大または最小	*VPMAX*、*VPMIN（整数）* P. A8-690
1011	0	0	–	上位半分を返すベクタ飽和乗算 2 倍	*VQDMULH* P. A8-704
		1	–	上位半分を返すベクタ飽和丸め付き乗算 2 倍	*VQRDMULH* P. A8-712
	1	0	–	ベクタペア単位加算	*VPADD（整数）* P. A8-684

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A7-9 Three registers of the same length (continued)

A	B	U	C	Instruction	See
1101	0	0	0x	Vector Add	*VADD (floating-point)* on page A8-538
			1x	Vector Subtract	*VSUB (floating-point)* on page A8-790
		1	0x	Vector Pairwise Add	*VPADD (floating-point)* on page A8-686
			1x	Vector Absolute Difference	*VABD (floating-point)* on page A8-530
	1	0	-	Vector Multiply Accumulate or Subtract	*VMLA, VMLS (floating-point)* on page A8-636
		1	0x	Vector Multiply	*VMUL (floating-point)* on page A8-664
1110	0	0	0x	Vector Compare Equal	*VCEQ (register)* on page A8-552
		1	0x	Vector Compare Greater Than or Equal	*VCGE (register)* on page A8-556
			1x	Vector Compare Greater Than	*VCGT (register)* on page A8-560
	1	1	-	Vector Absolute Compare Greater or Less Than (or Equal)	*VACGE, VACGT, VACLE,VACLT* on page A8-534
1111	0	0	-	Vector Maximum or Minimum	*VMAX, VMIN (floating-point)* on page A8-632
		1	-	Vector Pairwise Maximum or Minimum	*VPMAX, VPMIN (floating-point)* on page A8-692
	1	0	0x	Vector Reciprocal Step	*VRECPS* on page A8-730
		0	1x	Vector Reciprocal Square Root Step	*VRSQRTS* on page A8-744

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A7-9 同じ長さの 3 つのレジスタ（続き）

A	B	U	C	命令	参照先
1101	0	0	0x	ベクタ加算	*VADD（浮動小数点）* P. A8-538
			1x	ベクタ減算	*VSUB（浮動小数点数）* P. A8-790
		1	0x	ベクタペア単位加算	*VPADD（浮動小数点数）* P. A8-686
			1x	ベクタ絶対値の差	*VABD（浮動小数点）* P. A8-530
	1	0	–	ベクタ積和または積減算	*VMLA*、*VMLS（浮動小数点数）* P. A8-636
		1	0x	ベクタ乗算	*VMUL*、*VMULL（浮動小数点数）* P. A8-664
1110	0	0	0x	ベクタ比較等価	*VCEQ（レジスタ）* P. A8-552
		1	0x	ベクタ比較以上	*VCGE（レジスタ）* P. A8-556
			1x	ベクタ比較より大きい	*VCGT（レジスタ）* P. A8-560
	1	1	–	ベクタ絶対値比較より大きい またはより小さい（または等価）	*VACGE*、*VACGT*、*VACLE*、*VACLT* P. A8-534
1111	0	0	–	ベクタ最大または最小	*VMAX*、*VMIN（浮動小数点数）* P. A8-632
		1	–	ベクタペア単位最大 または最小	*VPMAX*、*VPMIN（浮動小数点数）* P. A8-692
	1	0	0x	ベクタ逆数ステップ	*VRECPS* P. A8-730
		0	1x	ベクタ逆数平方根ステップ	*VRSQRTS* P. A8-744

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4.2 Three registers of different lengths

Thumb encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1		B										A					0		0				

ARM encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1		B										A					0		0				

If B == 0b11, see *Advanced SIMD data-processing instructions* on page A7-10.

Table A7-10 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A7-10 Data-processing instructions with three registers of different lengths

A	U	Instruction	See
000x	-	Vector Add Long or Wide	*VADDL, VADDW* on page A8-542
001x	-	Vector Subtract Long or Wide	*VSUBL, VSUBW* on page A8-794
0100	0	Vector Add and Narrow, returning High Half	*VADDHN* on page A8-540
	1	Vector Rounding Add and Narrow, returning High Half	*VRADDHN* on page A8-726
0101	-	Vector Absolute Difference and Accumulate	*VABA, VABAL* on page A8-526
0110	0	Vector Subtract and Narrow, returning High Half	*VSUBHN* on page A8-792
	1	Vector Rounding Subtract and Narrow, returning High Half	*VRSUBHN* on page A8-748
0111	-	Vector Absolute Difference	*VABD, VABDL (integer)* on page A8-528
10x0	-	Vector Multiply Accumulate or Subtract	*VMLA, VMLAL, VMLS, VMLSL (integer)* on page A8-634
10x1	0	Vector Saturating Doubling Multiply Accumulate or Subtract Long	*VQDMLAL, VQDMLSL* on page A8-702
1100	-	Vector Multiply (integer)	*VMUL, VMULL (integer and polynomial)* on page A8-662
1101	0	Vector Saturating Doubling Multiply Long	*VQDMULL* on page A8-706
1110	-	Vector Multiply (polynomial)	*VMUL, VMULL (integer and polynomial)* on page A8-662

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4.2 異なる長さの 3 つのレジスタ

Thumb エンコード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1		B										A					0		0				

ARM エンコード

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1		B										A					0		0				

B == 0b11 の場合、*アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

この空間でのエンコードの割り当てを、表 A7-10 に示します。この空間に含まれる他のエンコードは未定義です。

表 A7-10 長さの異なる 3 つのレジスタを持つデータ処理命令

A	U	命令	参照先
000x	–	ベクタロング加算またはベクタワイド加算	*VADDL*、*VADDW* P. A8-542
001x	–	ベクタロング減算またはベクタワイド減算	*VSUBL*、*VSUBW* P. A8-794
0100	0	上位半分を返すベクタ加算および縮小	*VADDHN* P. A8-540
	1	上位半分を返すベクタ丸め加算および縮小	*VRADDHN* P. A8-726
0101	–	ベクタ絶対値の差および累積	*VABA*、*VABAL* P. A8-526
0110	0	上位半分を返すベクタ減算および縮小	*VSUBHN* P. A8-792
	1	上位半分を返すベクタ丸め減算および縮小	*VRSUBHN* P. A8-748
0111	–	ベクタ絶対値の差	*VABD*、*VABDL（整数）* P. A8-528
10x0	–	ベクタ積和または積減算	*VMLA*、*VMLAL*、*VMLS*、*VMLSL (整数)* P. A8-634
10x1	0	ベクタ飽和ロング積和または積減算 2 倍	*VQDMLAL*、*VQDMLSL* P. A8-702
1100	–	ベクタ乗算（整数）	*VMUL*、*VMULL（整数および多項式）* P. A8-662
1101	0	ベクタ飽和ロング乗算 2 倍	*VQDMULL* P. A8-706
1110	–	ベクタ乗算（多項式）	*VMUL*、*VMULL（整数および多項式）* P. A8-662

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4.3 Two registers and a scalar

Thumb encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1		B										A					1		0				

ARM encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1		B										A					1		0				

If B == 0b11, see *Advanced SIMD data-processing instructions* on page A7-10.

Table A7-11 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A7-11 Data-processing instructions with two registers and a scalar

A	U	Instruction	See
0x0x	-	Vector Multiply Accumulate or Subtract	*VMLA, VMLAL, VMLS, VMLSL (by scalar)* on page A8-638
0x10	-	Vector Multiply Accumulate or Subtract Long	*VMLA, VMLAL, VMLS, VMLSL (by scalar)* on page A8-638
0x11	0	Vector Saturating Doubling Multiply Accumulate or Subtract Long	*VQDMLAL, VQDMLSL* on page A8-702
100x	-	Vector Multiply	*VMUL, VMULL (by scalar)* on page A8-666
1010	-	Vector Multiply Long	*VMUL, VMULL (by scalar)* on page A8-666
1011	0	Vector Saturating Doubling Multiply Long	*VQDMULL* on page A8-706
1100	-	Vector Saturating Doubling Multiply returning High Half	*VQDMULH* on page A8-704
1101	-	Vector Saturating Rounding Doubling Multiply returning High Half	*VQRDMULH* on page A8-712

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4.3 2 つのレジスタと 1 つのスカラ

Thumb エンコード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1		B										A					1		0				

ARM エンコード

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1		B										A					1		0				

B == 0b11 の場合、*アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

この空間でのエンコードの割り当てを、表 A7-11 に示します。この空間に含まれる他のエンコードは未定義です。

表 A7-11 2 つのレジスタと 1 つのスカラを持つデータ処理命令

A	U	命令	参照先
0x0x	–	ベクタ積和演算または積減算	*VMLA*、*VMLAL*、*VMLS*、*VMLSL*（*スカラ*）P. A8-638
0x10	–	ベクタロング積和演算または積減算	*VMLA*、*VMLAL*、*VMLS*、*VMLSL*（*スカラ*）P. A8-638
0x11	0	ベクタ飽和ロング積和演算または積減算 2 倍	*VQDMLAL*、*VQDMLSL* P. A8-702
100x	–	ベクタ乗算	*VMUL*、*VMULL*（*スカラ*）P. A8-666
1010	–	ベクタロング乗算	*VMUL*、*VMULL*（*スカラ*）P. A8-666
1011	0	ベクタ飽和ロング乗算 2 倍	*VQDMULL* P. A8-706
1100	–	上位半分を返すベクタ飽和乗算 2 倍	*VQDMULH* P. A8-704
1101	–	上位半分を返すベクタ飽和丸め付き乗算 2 倍	*VQRDMULH* P. A8-712

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4.4 Two registers and a shift amount

Thumb encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1		imm3										A				L	B		1				

ARM encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1		imm3										A				L	B		1				

If [L, imm3] == 0b0000, see *One register and a modified immediate value* on page A7-21.

Table A7-12 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A7-12 Data-processing instructions with two registers and a shift amount

A	U	B	L	Instruction	See
0000	-	-	-	Vector Shift Right	*VSHR* on page A8-756
0001	-	-	-	Vector Shift Right and Accumulate	*VSRA* on page A8-764
0010	-	-	-	Vector Rounding Shift Right	*VRSHR* on page A8-738
0011	-	-	-	Vector Rounding Shift Right and Accumulate	*VRSRA* on page A8-746
0100	1	-	-	Vector Shift Right and Insert	*VSRI* on page A8-766
0101	0	-	-	Vector Shift Left	*VSHL (immediate)* on page A8-750
0101	1	-	-	Vector Shift Left and Insert	*VSLI* on page A8-760
011x	-	-	-	Vector Saturating Shift Left	*VQSHL, VQSHLU (immediate)* on page A8-720
1000	0	0	0	Vector Shift Right Narrow	*VSHRN* on page A8-758
		1	-	Vector Rounding Shift Right Narrow	*VRSHRN* on page A8-740
	1	0	-	Vector Saturating Shift Right, Unsigned Narrow	*VQSHRN, VQSHRUN* on page A8-722
		1	-	Vector Saturating Shift Right, Rounded Unsigned Narrow	*VQRSHRN, VQRSHRUN* on page A8-716
1001	-	0	-	Vector Saturating Shift Right, Narrow	*VQSHRN, VQSHRUN* on page A8-722
		1	-	Vector Saturating Shift Right, Rounded Narrow	*VQRSHRN, VQRSHRUN* on page A8-716

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4.4 2 つのレジスタとシフト量

Thumb エンコード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1		imm3										A				L	B		1				

ARM エンコード

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1		imm3										A				L	B		1				

[L、imm3] == 0b0000 の場合、*1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

この空間でのエンコードの割り当てを、表 A7-12 に示します。この空間に含まれる他のエンコードは未定義です。

表 A7-12 2 つのレジスタとシフト量を持つデータ処理命令

A	U	B	L	命令	参照先
0000	–	–	–	ベクタ右シフト	*VSHR* P. A8-756
0001	–	–	–	ベクタ右シフトおよび累積	*VSRA* P. A8-764
0010	–	–	–	ベクタ丸め付き右シフト	*VRSHR* P. A8-738
0011	–	–	–	ベクタ丸め付き右シフトおよび累積	*VRSRA* P. A8-746
0100	1	–	–	ベクタ右シフトおよび挿入	*VSRI* P. A8-766
0101	0	–	–	ベクタ左シフト	*VSHL（イミディエート）* P. A8-750
0101	1	–	–	ベクタ左シフトおよび挿入	*VSLI* P. A8-760
011x	–	–	–	ベクタ飽和左シフト	*VQSHL*、*VQSHLU（イミディエート）* P. A8-720
1000	0	0	0	ベクタ右シフトおよび縮小	*VSHRN* P. A8-758
		1	–	ベクタ丸め付き右シフトおよび縮小	*VRSHRN* P. A8-740
	1	0	–	ベクタ飽和右シフト、符号なし縮小	*VQSHRN*、*VQSHRUN* P. A8-722
		1	–	ベクタ飽和右シフト、丸め付き符号なし縮小	*VQRSHRN*、*VQRSHRUN* P. A8-716
1001	–	0	–	ベクタ飽和右シフト、縮小	*VQSHRN*、*VQSHRUN* P. A8-722
		1	–	ベクタ飽和右シフト、丸め付き縮小	*VQRSHRN*、*VQRSHRUN* P. A8-716

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A7-12 Data-processing instructions with two registers and a shift amount (continued)

A	U	B	L	Instruction	See
1010	-	0	-	Vector Shift Left Long	*VSHLL* on page A8-754
				Vector Move Long	*VMOVL* on page A8-654
111x	-	-	-	Vector Convert	*VCVT (between floating-point and fixed-point, Advanced SIMD)* on page A8-580

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A7-12 2 つのレジスタとシフト量を持つデータ処理命令（続き）

A	U	B	L	命令	参照先
1010	–	0	–	ベクタ左シフトロング	*VSHLL* P. A8-754
				ベクタロング移動	*VMOVL* P. A8-654
111x	–	–	–	ベクタ変換	*VCVT（浮動小数点数と固定小数点数との間、アドバンスト SIMD）* P. A8-580

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4.5 Two registers, miscellaneous

Thumb encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1		1	1			A						0	B						0				

ARM encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1		1	1			A						0	B						0				

The allocation of encodings in this space is shown in Table A7-13. Other encodings in this space are UNDEFINED.

Table A7-13 Instructions with two registers, miscellaneous

A	B	Instruction	See
00	0000x	Vector Reverse in doublewords	*VREV16, VREV32, VREV64* on page A8-732
	0001x	Vector Reverse in words	*VREV16, VREV32, VREV64* on page A8-732
	0010x	Vector Reverse in halfwords	*VREV16, VREV32, VREV64* on page A8-732
	010xx	Vector Pairwise Add Long	*VPADDL* on page A8-688
	1000x	Vector Count Leading Sign Bits	*VCLS* on page A8-566
	1001x	Vector Count Leading Zeros	*VCLZ* on page A8-570
	1010x	Vector Count	*VCNT* on page A8-574
	1011x	Vector Bitwise NOT	*VMVN (register)* on page A8-670
	110xx	Vector Pairwise Add and Accumulate Long	*VPADAL* on page A8-682
	1110x	Vector Saturating Absolute	*VQABS* on page A8-698
	1111x	Vector Saturating Negate	*VQNEG* on page A8-710

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4.5 2 つのレジスタ、その他

Thumb エンコード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1		1	1			A						0	B						0				

ARM エンコード

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1		1	1			A						0	B						0				

この空間におけるエンコードの割り当てを、表 A7-13 に示します。この空間に含まれる他のエンコードは未定義です。

表 A7-13 2 つのレジスタを持つその他の命令

A	B	命令	参照先
00	0000x	ダブルワードでのベクタ反転	*VREV16*、*VREV32*、*VREV64* P. A8-732
	0001x	ワードでのベクタ反転	*VREV16*、*VREV32*、*VREV64* P. A8-732
	0010x	ハーフワードでのベクタ反転	*VREV16*、*VREV32*、*VREV64* P. A8-732
	010xx	ベクタペア単位ロング加算	*VPADDL* P. A8-688
	1000x	ベクタ先頭符号ビットカウント	*VCLS* P. A8-566
	1001x	ベクタ先行ゼロカウント	*VCLZ* P. A8-570
	1010x	ベクタカウント	*VCNT* P. A8-574
	1011x	ベクタビット単位否定	*VMVN（レジスタ）* P. A8-670
	110xx	ベクタペア単位ロング加算および累積	*VPADAL* P. A8-682
	1110x	ベクタ飽和絶対値	*VQABS* P. A8-698
	1111x	ベクタ飽和否定	*VQNEG* P. A8-710

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A7-13 Instructions with two registers, miscellaneous (continued)

A	B	Instruction	See
01	x000x	Vector Compare Greater Than Zero	*VCGT (immediate #0)* on page A8-562
	x001x	Vector Compare Greater Than or Equal to Zero	*VCGE (immediate #0)* on page A8-558
	x010x	Vector Compare Equal to zero	*VCEQ (immediate #0)* on page A8-554
	x011x	Vector Compare Less Than or Equal to Zero	*VCLE (immediate #0)* on page A8-564
	x100x	Vector Compare Less Than Zero	*VCLT (immediate #0)* on page A8-568
	x110x	Vector Absolute	*VABS* on page A8-532
	x111x	Vector Negate	*VNEG* on page A8-672
	0000x	Vector Swap	*VSWP* on page A8-796
	0001x	Vector Transpose	*VTRN* on page A8-800
	0010x	Vector Unzip	*VUZP* on page A8-804
	0011x	Vector Zip	*VZIP* on page A8-806
10	01000	Vector Move and Narrow	*VMOVN* on page A8-656
	01001	Vector Saturating Move and Unsigned Narrow	*VQMOVN, VQMOVUN* on page A8-708
	0101x	Vector Saturating Move and Narrow	*VQMOVN, VQMOVUN* on page A8-708
	01100	Vector Shift Left Long (maximum shift)	*VSHLL* on page A8-754
	11x00	Vector Convert	*VCVT (between half-precision and single-precision, Advanced SIMD)* on page A8-586
11	10x0x	Vector Reciprocal Estimate	*VRECPE* on page A8-728
	10x1x	Vector Reciprocal Square Root Estimate	*VRSQRTE* on page A8-742
	11xxx	Vector Convert	*VCVT (between floating-point and integer, Advanced SIMD)* on page A8-576

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A7-13 2 つのレジスタを持つその他の命令（続き）

A	B	命令	参照先
01	x000x	ベクタ比較ゼロより大きい	*VCGT（イミディエート #0）* P. A8-562
	x001x	ベクタ比較ゼロ以上	*VCGE（イミディエート #0）* P. A8-558
	x010x	ベクタ比較ゼロに等価	*VCEQ（イミディエート #0）* P. A8-554
	x011x	ベクタ比較ゼロ以下	*VCLE（イミディエート #0）* P. A8-564
	x100x	ベクタ比較ゼロより小さい	*VCLT（イミディエート #0）* P. A8-568
	x110x	ベクタ絶対値	*VABS* P. A8-532
	x111x	ベクタ否定	*VNEG* P. A8-672
	0000x	ベクタスワップ	*VSWP* P. A8-796
	0001x	ベクタ転置	*VTRN* P. A8-800
	0010x	ベクタアンジップ	*VUZP* P. A8-804
	0011x	ベクタジップ	*VZIP* P. A8-806
10	01000	ベクタ移動および縮小	*VMOVN* P. A8-656
	01001	ベクタ飽和移動および符号なし縮小	*VQMOVN*、*VQMOVUN* P. A8-708
	0101x	ベクタ飽和移動および縮小	*VQMOVN*、*VQMOVUN* P. A8-708
	01100	ベクタ左シフトロング（最大シフト）	*VSHLL* P. A8-754
	11x00	ベクタ変換	*VCVT（半精度と単精度との間、アドバンスト SIMD）* P. A8-586
11	10x0x	ベクタ逆数概算	*VRECPE* P. A8-728
	10x1x	ベクタ逆数平方根概算	*VRSQRTE* P. A8-742
	11xxx	ベクタ変換	*VCVT（浮動小数点数と整数との間、アドバンスト SIMD）* P. A8-576

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4.6 One register and a modified immediate value

Thumb encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	a	1	1	1	1	1		0	0	0	b	c	d					cmode				0		op	1	e	f	g	h

ARM encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	a	1		0	0	0	b	c	d					cmode				0		op	1	e	f	g	h

Table A7-14 shows the allocation of encodings in this space.

Table A7-15 on page A7-22 shows the modified immediate constants available with these instructions, and how they are encoded.

Table A7-14 Data-processing instructions with one register and a modified immediate value

op	cmode	Instruction	See
0	0xx0	Vector Move	*VMOV (immediate)* on page A8-640
	0xx1	Vector Bitwise OR	*VORR (immediate)* on page A8-678
	10x0	Vector Move	*VMOV (immediate)* on page A8-640
	10x1	Vector Bitwise OR	*VORR (immediate)* on page A8-678
	11xx	Vector Move	*VMOV (immediate)* on page A8-640
1	0xx0	Vector Bitwise NOT	*VMVN (immediate)* on page A8-668
	0xx1	Vector Bit Clear	*VBIC (immediate)* on page A8-546
	10x0	Vector Bitwise NOT	*VMVN (immediate)* on page A8-668
	10x1	Vector Bit Clear	*VBIC (immediate)* on page A8-546
	110x	Vector Bitwise NOT	*VMVN (immediate)* on page A8-668
	1110	Vector Move	*VMOV (immediate)* on page A8-640
	1111	UNDEFINED	-

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.4.6 1 つのレジスタと修飾イミディエート値

Thumb エンコード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	a	1	1	1	1	1		0	0	0	b	c	d					cmode				0		op	1	e	f	g	h

ARM エンコード

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	a	1		0	0	0	b	c	d					cmode				0		op	1	e	f	g	h

この空間でのエンコードの割り当てを、表 A7-14 に示します。

これらの命令で使用可能な修飾イミディエート定数とそのエンコードを、P. A7-22 の表 A7-15 に示します。

表 A7-14 1 つのレジスタと修飾イミディエート値を持つデータ処理命令

op	cmode	命令	参照先
0	0xx0	ベクタ移動	*VMOV（イミディエート）* P. A8-640
	0xx1	ベクタビット単位論理和	*VORR（イミディエート）* P. A8-678
	10x0	ベクタ移動	*VMOV（イミディエート）* P. A8-640
	10x1	ベクタビット単位論理和	*VORR（イミディエート）* P. A8-678
	11xx	ベクタ移動	*VMOV（イミディエート）* P. A8-640
1	0xx0	ベクタビット単位否定	*VMVN（イミディエート）* P. A8-668
	0xx1	ベクタ ビットクリア	*VBIC（イミディエート）* P. A8-546
	10x0	ベクタビット単位否定	*VMVN（イミディエート）* P. A8-668
	10x1	ベクタ ビットクリア	*VBIC（イミディエート）* P. A8-546
	110x	ベクタビット単位否定	*VMVN（イミディエート）* P. A8-668
	1110	ベクタ移動	*VMOV（イミディエート）* P. A8-640
	1111	未定義	–

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A7-15 Modified immediate values for Advanced SIMD instructions

op	cmode	Constant [a]	<dt>[b]	Notes
-	000x	00000000 00000000 00000000 abcdefgh 00000000 00000000 00000000 abcdefgh	I32	c
	001x	00000000 00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000	I32	c, d
	010x	00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000	I32	c, d
	011x	abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000 00000000	I32	c, d
	100x	00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh	I16	c
	101x	abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000	I16	c, d
	1100	00000000 00000000 abcdefgh 11111111 00000000 00000000 abcdefgh 11111111	I32	d, e
	1101	00000000 abcdefgh 11111111 11111111 00000000 abcdefgh 11111111 11111111	I32	d, e
0	1110	abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh	I8	f
1	1110	aaaaaaaa bbbbbbbb cccccccc dddddddd eeeeeeee ffffffff gggggggg hhhhhhhh	I64	f
0	1111	aBbbbbbc defgh000 00000000 00000000 aBbbbbbc defgh000 00000000 00000000	F32	f, g
1	1111	UNDEFINED	-	-

a. In this table, the immediate value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembler syntax, the constant is specified by a data type and a value of that type. That value is specified in the normal way (a decimal number by default) and is replicated enough times to fill the 64-bit immediate. For example, a data type of `I32` and a value of 10 specify the 64-bit constant `0x0000000A0000000A`.

b. This specifies the data type used when the instruction is disassembled. On assembly, the data type must be matched in the table if possible. Other data types are permitted as pseudo-instructions when code is assembled, provided the 64-bit constant specified by the data type and value is available for the instruction (if it is available in more than one way, the first entry in this table that can produce it is used). For example, `VMOV.I64 D0,#0x8000000080000000` does not specify a 64-bit constant that is available from the I64 line of the table, but does specify one that is available from the fourth I32 line or the F32 line. It is assembled to the former, and therefore is disassembled as `VMOV.I32 D0,#0x80000000`.

c. This constant is available for the `VBIC`, `VMOV`, `VMVN`, and `VORR` instructions.

d. `UNPREDICTABLE` if `abcdefgh` == 00000000.

e. This constant is available for the `VMOV` and `VMVN` instructions only.

f. This constant is available for the `VMOV` instruction only.

g. In this entry, `B` = NOT(`b`). The bit pattern represents the floating-point number $(-1)^S * 2^{exp} *$ mantissa, where S = `UInt(a)`, exp = `UInt(NOT(b):c:d)-3` and mantissa = `(16+UInt(e:f:g:h))/16`.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A7-15 アドバンスト SIMD 命令における修飾イミディエート値

op	cmode	定数[a]	<dt>[b]	注
–	000x	00000000 00000000 00000000 abcdefgh 00000000 00000000 00000000 abcdefgh	I32	c
	001x	00000000 00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000	I32	c, d
	010x	00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000	I32	c, d
	011x	abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000 00000000	I32	c, d
	100x	00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh	I16	c
	101x	abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000	I16	c, d
	1100	00000000 00000000 abcdefgh 11111111 00000000 00000000 abcdefgh 11111111	I32	d, e
	1101	00000000 abcdefgh 11111111 11111111 00000000 abcdefgh 11111111 11111111	I32	d, e
0	1110	abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh	I8	f
1	1110	aaaaaaaa bbbbbbbb cccccccc dddddddd eeeeeeee ffffffff gggggggg hhhhhhhh	I64	f
0	1111	aBbbbbbc defgh000 00000000 00000000 aBbbbbbc defgh000 00000000 00000000	F32	f, g
1	1111	未定義	–	–

a. この表では、abcdefgh をエンコード図に関連付けるため、イミディエート値をバイナリ形式で示しています。アセンブラ構文で、定数はデータ型とその型の値によって指定されます。その値は通常の方法（デフォルトでは 10 進数）で指定され、64 ビットのイミディエートを満たすのに必要な回数だけ複製されます。例えば、データ型 `I32` と値 10 は、64 ビットの定数 `0x0000000A0000000A` を指定します。

b. これは、命令の逆アセンブルで使用されるデータ型を指定します。アセンブリにおいて、データ型は可能であれば表内で一致している必要があります。他のデータ型は、コードがアセンブルされるとき、擬似命令として許容されます。ただし、データ型と値によって指定される 64 ビット定数がその命令で使用可能である必要があります（2 つ以上の形で使用可能である場合は、それを生成するこの表の最初のエントリが使用されます）。例えば、`VMOV.I64 D0,#0x8000000080000000` は、表の I64 行から使用可能な 64 ビットの定数を指定しませんが、4 番目の I32 行または F32 行から使用可能な定数を指定します。これは前者にアセンブルされるため、`VMOV.I32 D0,#0x80000000` として逆アセンブルされます。

c. この定数は、`VBIC`、`VMOV`、`VMVN`、`VORR` 命令に使用可能です。

d. `abcdefgh` == 00000000 の場合、予測不能です。

e. この定数は、`VMOV` および `VMVN` 命令にのみ使用可能です。

f. この定数は `VMOV` 命令にのみ使用可能です。

g. このエントリでは、`B` = NOT(`b`) です。このビットパターンは、浮動小数点数 $(-1)^S * 2^{exp} *$ 仮数を示します。このとき、S = `UInt(a)`、exp = `UInt(NOT(b):c:d) - 3` で、仮数 = `(16 + UInt(e:f:g:h)) / 16` です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Operation

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)

    case cmode<3:1> of
        when '000'
            testimm8 = FALSE;  imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            testimm8 = TRUE;   imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            testimm8 = TRUE;   imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            testimm8 = TRUE;   imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            testimm8 = FALSE;  imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            testimm8 = TRUE;   imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            testimm8 = TRUE;
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            testimm8 = FALSE;
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8);  imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8);  imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8);  imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8);  imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                UNDEFINED;

    if testimm8 && imm8 == '00000000' then
        UNPREDICTABLE;

    return imm64;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

動作

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)

    case cmode<3:1> of
        when '000'
            testimm8 = FALSE;  imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            testimm8 = TRUE;   imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            testimm8 = TRUE;   imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            testimm8 = TRUE;   imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            testimm8 = FALSE;  imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            testimm8 = TRUE;   imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            testimm8 = TRUE;
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            testimm8 = FALSE;
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8);  imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8);  imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8);  imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8);  imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                UNDEFINED;

    if testimm8 && imm8 == '00000000' then
        UNPREDICTABLE;

    return imm64;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.5 VFP data-processing instructions

Thumb encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	T	1	1	1	0	opc1				opc2								1	0	1		opc3			0	opc4			

ARM encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	opc1				opc2								1	0	1		opc3			0	opc4			

If `T` == 1 in the Thumb encoding or `cond` == 0b1111 in the ARM encoding, the instruction is UNDEFINED.

Otherwise:

- Table A7-16 shows the encodings for three-register VFP data-processing instructions. Other encodings in this space are UNDEFINED.
- Table A7-17 on page A7-25 applies only if Table A7-16 indicates that it does. It shows the encodings for VFP data-processing instructions with two registers or a register and an immediate. Other encodings in this space are UNDEFINED.
- Table A7-18 on page A7-25 shows the immediate constants available in the `VMOV` (immediate) instruction.

These instructions are `CDP` instructions for coprocessors 10 and 11.

Table A7-16 Three-register VFP data-processing instructions

opc1	opc3	Instruction	See
0x00	-	Vector Multiply Accumulate or Subtract	*VMLA, VMLS (floating-point)* on page A8-636
0x01	-	Vector Negate Multiply Accumulate or Subtract	*VNMLA, VNMLS, VNMUL* on page A8-674
0x10	x1		
	x0	Vector Multiply	*VMUL (floating-point)* on page A8-664
0x11	x0	Vector Add	*VADD (integer)* on page A8-536
	x1	Vector Subtract	*VSUB (integer)* on page A8-788
1x00	x0	Vector Divide	*VDIV* on page A8-590
1x11	-	Other VFP data-processing instructions	Table A7-17 on page A7-25

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.5 VFP のデータ処理命令

Thumb エンコード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	T	1	1	1	0	opc1				opc2								1	0	1		opc3			0	opc4			

ARM エンコード

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	opc1				opc2								1	0	1		opc3			0	opc4			

Thumb エンコードで `T` == 1 のとき、または ARM エンコードで `cond` == 0b1111 のとき、命令は未定義です。

それ以外の場合、詳細は次の表に示すとおりです。

- 3 レジスタの VFP データ処理命令のエンコードを、表 A7-16 に示します。この空間に含まれる他のエンコードは未定義です。
- P. A7-25 の表 A7-17 は、表 A7-16 でそのように記載されている場合のみ適用されます。この表は、2 つのレジスタまたは 1 つのレジスタとイミディエートを持つ VFP データ処理命令のエンコードを示しています。この空間に含まれる他のエンコードは未定義です。
- P. A7-25 の表 A7-18 は、`VMOV`（イミディエート）命令で使用可能なイミディエート定数を示しています。

これらの命令は、コプロセッサ 10 および 11 の `CDP` 命令です。

表 A7-16 3 レジスタの VFP データ処理命令

opc1	opc3	命令	参照先
0x00	–	ベクタ積和演算または積減算	*VMLA*、*VMLS（浮動小数点数）* P. A8-636
0x01	–	ベクタ否定積和演算または積減算	*VNMLA*、*VNMLS*、*VNMUL* P. A8-674
0x10	x1		
	x0	ベクタ乗算	*VMUL*、*VMULL（浮動小数点数）* P. A8-664
0x11	x0	ベクタ加算	*VADD（整数）* P. A8-536
	x1	ベクタ減算	*VSUB（整数）* P. A8-788
1x00	x0	ベクタ除算	*VDIV* P. A8-590
1x11	–	その他の VFP データ処理命令	P. A7-25 の表 A7-17

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A7-17 Other VFP data-processing instructions

opc2	opc3	Instruction	See
-	x0	Vector Move	*VMOV (immediate)* on page A8-640
0000	01	Vector Move	*VMOV (register)* on page A8-642
	11	Vector Absolute	*VABS* on page A8-532
0001	01	Vector Negate	*VNEG* on page A8-672
	11	Vector Square Root	*VSQRT* on page A8-762
001x	x1	Vector Convert	*VCVTB, VCVTT (between half-precision and single-precision, VFP)* on page A8-588
010x	x1	Vector Compare	*VCMP, VCMPE* on page A8-572
0111	11	Vector Convert	*VCVT (between double-precision and single-precision)* on page A8-584
1000	x1	Vector Convert	*VCVT, VCVTR (between floating-point and integer, VFP)* on page A8-578
101x	x1	Vector Convert	*VCVT (between floating-point and fixed-point, VFP)* on page A8-582
110x	x1	Vector Convert	*VCVT, VCVTR (between floating-point and integer, VFP)* on page A8-578
111x	x1	Vector Convert	*VCVT (between floating-point and fixed-point, VFP)* on page A8-582

Table A7-18 VFP modified immediate constants

Data type	opc2	opc4	Constant [a]
F32	abcd	efgh	aBbbbbbc defgh000 00000000 00000000
F64	abcd	efgh	aBbbbbbb bbcdefgh 00000000 00000000 00000000 00000000 00000000 00000000

a. In this column, B = NOT(b). The bit pattern represents the floating-point number $(-1)^S * 2^{exp} *$ mantissa, where S = UInt(a), exp = UInt(NOT(b):c:d)-3 and mantissa = (16+UInt(e:f:g:h))/16.

A7.5.1 Operation

```
// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8, integer N)
    assert N == 32 || N == 64;
    if N == 32 then
        return imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
    else
        return imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A7-17 その他の VFP データ処理命令

opc2	opc3	命令	参照先
-	x0	ベクタ移動	*VMOV（イミディエート）* P. A8-640
0000	01	ベクタ移動	*VMOV（レジスタ）* P. A8-642
	11	ベクタ絶対値	*VABS* P. A8-532
0001	01	ベクタ否定	*VNEG* P. A8-672
	11	ベクタ平方根	*VSQRT* P. A8-762
001x	x1	ベクタ変換	*VCVTB、VCVTT（半精度と単精度との間、VFP）* P. A8-588
010x	x1	ベクタ比較	*VCMP、VCMPE* P. A8-572
0111	11	ベクタ変換	*VCVT（倍精度と単精度との間）* P. A8-584
1000	x1	ベクタ変換	*VCVT、VCVTR（浮動小数点数と整数との間、VFP）* P. A8-578
101x	x1	ベクタ変換	*VCVT（浮動小数点数と固定小数点数との間、VFP）* P. A8-582
110x	x1	ベクタ変換	*VCVT、VCVTR（浮動小数点数と整数との間、VFP）* P. A8-578
111x	x1	ベクタ変換	*VCVT（浮動小数点数と固定小数点数との間、VFP）* P. A8-582

表 A7-18 ベクタ修飾イミディエート定数

データ型	opc2	opc4	定数[a]
F32	abcd	efgh	aBbbbbbc defgh000 00000000 00000000
F64	abcd	efgh	aBbbbbbb bbcdefgh 00000000 00000000 00000000 00000000 00000000 00000000

a. この列では、B = NOT(b) です。このビットパターンは、浮動小数点数 $(-1)^S * 2^{exp} *$ 仮数を示します。このとき、S = UInt(a)、exp = UInt(NOT(b):c:d) - 3 で、仮数 = (16 + UInt(e:f:g:h)) / 16 です。

A7.5.1 動作

```
// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8, integer N)
    assert N == 32 || N == 64;
    if N == 32 then
        return imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
    else
        return imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.6 Extension register load/store instructions

Thumb encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	T	1	1	0	Opcode					Rn								1	0	1									

ARM encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	Opcode					Rn								1	0	1									

If `T` == 1 in the Thumb encoding or `cond` == 0b1111 in the ARM encoding, the instruction is UNDEFINED.

Otherwise, the allocation of encodings in this space is shown in Table A7-19. Other encodings in this space are UNDEFINED.

These instructions are `LDC` and `STC` instructions for coprocessors 10 and 11.

Table A7-19 Extension register load/store instructions

Opcode	Rn	Instruction	See
0010x	-	-	*64-bit transfers between ARM core and extension registers* on page A7-32
01x00	-	Vector Store Multiple (Increment After, no writeback)	*VSTM* on page A8-784
01x10	-	Vector Store Multiple (Increment After, writeback)	*VSTM* on page A8-784
1xx00	-	Vector Store Register	*VSTR* on page A8-786
10x10	not 1101	Vector Store Multiple (Decrement Before, writeback)	*VSTM* on page A8-784
	1101	Vector Push Registers	*VPUSH* on page A8-696
01x01	-	Vector Load Multiple (Increment After, no writeback)	*VLDM* on page A8-626
01x11	not 1101	Vector Load Multiple (Increment After, writeback)	*VLDM* on page A8-626
	1101	Vector Pop Registers	*VPOP* on page A8-694
1xx01	-	Vector Load Register	*VLDR* on page A8-628
10x11	-	Vector Load Multiple (Decrement Before, writeback)	*VLDM* on page A8-626

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.6 拡張機能レジスタロード / ストア命令

Thumb エンコード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	T	1	1	0	オペコード					Rn								1	0	1									

ARM エンコード

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	オペコード					Rn								1	0	1									

Thumb エンコードで `T` == 1 のとき、または ARM エンコードで `cond` == 0b1111 のとき、命令は未定義です。

それ以外の場合、この空間におけるエンコードの割り当てを表 A7-19 に示します。この空間に含まれる他のエンコードは未定義です。

これらの命令は、コプロセッサ 10、11 の `LDC` および `STC` 命令です。

表 A7-19 拡張機能レジスタロード / ストア命令

オペコード	Rn	命令	参照先
0010x	–	–	*ARM コアレジスタと拡張機能レジスタとの 64 ビット転送* P. A7-32
01x00	–	ベクタ複数ストア（ポストインクリメント、ライトバックなし）	*VSTM* P. A8-784
01x10	–	ベクタ複数ストア（ポストインクリメント、ライトバック）	*VSTM* P. A8-784
1xx00	–	ベクタ レジスタストア	*VSTR* P. A8-786
10x10	1101 以外	ベクタ複数ストア（プリデクリメント、ライトバック）	*VSTM* P. A8-784
	1101	ベクタ レジスタプッシュ	*VPUSH* P. A8-696
01x01	–	ベクタ複数ロード（ポストインクリメント、ライトバックなし）	*VLDM* P. A8-626
01x11	1101 以外	ベクタ複数ロード（ポストインクリメント、ライトバック）	*VLDM* P. A8-626
	1101	ベクタ レジスタポップ	*VPOP* P. A8-694
1xx01	–	ベクタ レジスタロード	*VLDR* P. A8-628
10x11	–	ベクタ複数ロード（プリデクリメント、ライトバック）	*VLDM* P. A8-626

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.7 Advanced SIMD element or structure load/store instructions

Thumb encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	A		L	0									B											

ARM encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	A		L	0									B											

The allocation of encodings in this space is shown in:

- Table A7-20 if L == 0, store instructions
- Table A7-21 on page A7-28 if L == 1, load instructions.

Other encodings in this space are UNDEFINED.

The variable bits are in identical locations in the two encodings, after adjusting for the fact that the ARM encoding is held in memory as a single word and the Thumb encoding is held as two consecutive halfwords.

The ARM instructions can only executed unconditionally. The Thumb instructions can be executed conditionally by using the IT instruction. For details see *IT* on page A8-104.

Table A7-20 Element and structure store instructions (L == 0)

A	B	Instruction	See
0	0010 011x 1010	Vector Store	*VST1 (multiple single elements)* on page A8-768
	0011 100x	Vector Store	*VST2 (multiple 2-element structures)* on page A8-772
	010x	Vector Store	*VST3 (multiple 3-element structures)* on page A8-776
	000x	Vector Store	*VST4 (multiple 4-element structures)* on page A8-780

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.7 アドバンスト SIMD のエレメントまたは構造ロード / ストア命令

Thumb エンコード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	A		L	0									B											

ARM エンコード

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	A		L	0									B											

この空間におけるエンコードの割り当てを、次の表に示します。

- L == 0 の場合表 A7-20、ストア命令
- L == 1 の場合 P. A7-28 の表 A7-21、ロード命令

この空間に含まれる他のエンコードは未定義です。

可変ビットは 2 つのエンコードで同じ位置にあります。ただし、ARM エンコードは単一ワードとしてメモリに保持され、Thumb エンコードは 2 つの連続するハーフワードとして保持される点が異なります。

ARM 命令は無条件でしか実行できません。Thumb 命令は IT 命令を使用して条件付き実行が可能です。詳細については、*IT* P. A8-104 を参照して下さい。

表 A7-20 エレメントおよび構造ストア命令 (L == 0)

A	B	命令	参照先
0	0010 011x 1010	ベクタストア	*VST1（複数の単一エレメントのストア）* P. A8-768
	0011 100x	ベクタストア	*VST2（複数の 2 エレメント構造のストア）* P. A8-772
	010x	ベクタストア	*VST3（複数の 3 エレメント構造のストア）* P. A8-776
	000x	ベクタストア	*VST4（複数の 4 エレメント構造のストア）* P. A8-780

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A7-20 Element and structure store instructions (L == 0) (continued)

A	B	Instruction	See
1	0x00 1000	Vector Store	*VST1 (single element from one lane)* on page A8-770
	0x01 1001	Vector Store	*VST2 (single 2-element structure from one lane)* on page A8-774
	0x10 1010	Vector Store	*VST3 (single 3-element structure from one lane)* on page A8-778
	0x11 1011	Vector Store	*VST4 (single 4-element structure from one lane)* on page A8-782

Table A7-21 Element and structure load instructions (L == 1)

A	B	Instruction	See
0	0010 011x 1010	Vector Load	*VLD1 (multiple single elements)* on page A8-602
	0011 100x	Vector Load	*VLD2 (multiple 2-element structures)* on page A8-608
	010x	Vector Load	*VLD3 (multiple 3-element structures)* on page A8-614
	000x	Vector Load	*VLD4 (multiple 4-element structures)* on page A8-620

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A7-20 エレメントおよび構造ストア命令 (L = 0)（続き）

A	B	命令	参照先
1	0x00 1000	ベクタストア	*VST1（1 つのレーンからの単一エレメントのストア）* P. A8-770
	0x01 1001	ベクタストア	*VST2（1 つのレーンからの単一の 2 エレメント構造のストア）* P. A8-774
	0x10 1010	ベクタストア	*VST3（1 つのレーンからの単一の 3 エレメント構造のストア）* P. A8-778
	0x11 1011	ベクタストア	*VST4（1 つのレーンからの単一の 4 エレメント構造のストア）* P. A8-782

表 A7-21 エレメントおよび構造ロード命令 (L = = 1)

A	B	命令	参照先
0	0010 011x 1010	ベクタロード	*VLD1（複数の単一エレメントのロード）* P. A8-602
	0011 100x	ベクタロード	*VLD2（複数の 2 エレメント構造のロード）* P. A8-608
	010x	ベクタロード	*VLD3（複数の 3 エレメント構造のロード）* P. A8-614
	000x	ベクタロード	*VLD4（複数の 4 エレメント構造のロード）* P. A8-620

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A7-21 Element and structure load instructions (L == 1) (continued)

A	B	Instruction	See
1	0x00 1000	Vector Load	*VLD1 (single element to one lane)* on page A8-604
	1100	Vector Load	*VLD1 (single element to all lanes)* on page A8-606
	0x01 1001	Vector Load	*VLD2 (single 2-element structure to one lane)* on page A8-610
	1101	Vector Load	*VLD2 (single 2-element structure to all lanes)* on page A8-612
	0x10 1010	Vector Load	*VLD3 (single 3-element structure to one lane)* on page A8-616
	1110	Vector Load	*VLD3 (single 3-element structure to all lanes)* on page A8-618
	0x11 1011	Vector Load	*VLD4 (single 4-element structure to one lane)* on page A8-622
	1111	Vector Load	*VLD4 (single 4-element structure to all lanes)* on page A8-624

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A7-21 エレメントおよび構造ロード命令（L = = 1）（続き）

A	B	命令	参照先
1	0x00 1000	ベクタロード	*VLD1（1 つのレーンへの単一エレメントのロード）* P. A8-604
	1100	ベクタロード	*VLD1（全レーンへの単一エレメントのロード）* P. A8-606
	0x01 1001	ベクタロード	*VLD2（1 つのレーンへの単一の 2 エレメント構造のロード）* P. A8-610
	1101	ベクタロード	*VLD2（全レーンへの単一の 2 エレメント構造のロード）* P. A8-612
	0x10 1010	ベクタロード	*VLD3（1 つのレーンへの単一の 3 エレメント構造のロード）* P. A8-616
	1110	ベクタロード	*VLD3（全レーンへの単一の 3 エレメント構造のロード）* P. A8-618
	0x11 1011	ベクタロード	*VLD4（1 つのレーンへの単一の 4 エレメント構造のロード）* P. A8-622
	1111	ベクタロード	*VLD4（全レーンへの単一の 4 エレメント構造のロード）* P. A8-624

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.7.1 Advanced SIMD addressing mode

All the element and structure load/store instructions use this addressing mode. There is a choice of three formats:

`[<Rn>{@<align>}]`

The address is contained in ARM core register Rn.

Rn is not updated by this instruction.

Encoded as Rm = 0b1111.

If Rn is encoded as 0b1111, the instruction is UNPREDICTABLE.

`[<Rn>{@<align>}]!`

The address is contained in ARM core register Rn.

Rn is updated by this instruction: `Rn = Rn + transfer_size`

Encoded as Rm = 0b1101.

`transfer_size` is the number of bytes transferred by the instruction. This means that, after the instruction is executed, Rn points to the address in memory immediately following the last address loaded from or stored to.

If Rn is encoded as 0b1111, the instruction is UNPREDICTABLE.

This addressing mode can also be written as:

`[<Rn>{@align}], #<transfer_size>`

However, disassembly produces the `[<Rn>{@align}]!` form.

`[<Rn>{@<align>}], <Rm>`

The address is contained in ARM core register `<Rn>`.

Rn is updated by this instruction: `Rn = Rn + Rm`

Encoded as Rm = Rm. Rm must not be encoded as 0b1111 or 0b1101 (the PC or the SP).

If Rn is encoded as 0b1111, the instruction is UNPREDICTABLE.

In all cases, `<align>` specifies an optional alignment. Details are given in the individual instruction descriptions.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.7.1 アドバンスト SIMD のアドレシングモード

エレメントおよび構造のロード / ストア命令は、すべてこのアドレシングモードを使用します。フォーマットは 3 つのうちから選択できます。

`[<Rn>{@<align>}]`

このアドレスは ARM コアレジスタ Rn に含まれます。

Rn はこの命令によって更新されません。

Rm = 0b1111 としてエンコードされます。

Rn が 0b1111 としてエンコードされている場合、命令は予測不能です。

`[<Rn>{@<align>}]!`

このアドレスは ARM コアレジスタ Rn に含まれます。

Rn はこの命令によって更新されます。`Rn = Rn + transfer_size`

Rm = 0b1101 としてエンコードされます。

`transfer_size` は、命令によって転送されるバイトの数です。つまり、命令が実行された後、Rn はメモリ内において最後のロード元またはストア先のアドレスの直後のアドレスを指すことになります。

Rn が 0b1111 としてエンコードされている場合、命令は予測不能です。

このアドレシングモードは、次のように記述することもできます。

`[<Rn>{@align}], #<transfer_size>`

ただし、逆アセンブリでは `[<Rn>{@align}]!` の形式が生成されます。

`[<Rn>{@<align>}], <Rm>`

このアドレスは ARM コアレジスタの `<Rn>` に含まれます。

Rn はこの命令によって更新されます。`Rn = Rn + Rm`

Rm = Rm としてエンコードされます。Rm を 0b1111 または 0b1101（PC または SP）としてエンコードすることはできません。

Rn が 0b1111 としてエンコードされている場合、命令は予測不能です。

どのような場合でも、`<align>` は常にオプションのアライメントを指定します。詳細は、各命令の説明に記載されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.8 8, 16, and 32-bit transfer between ARM core and extension registers

Thumb encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	T	1	1	1	0	A			L									1	0	1	C		B			1			

ARM encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	A			L									1	0	1	C		B			1			

If `T` == 1 in the Thumb encoding or `cond` == 0b1111 in the ARM encoding, the instruction is UNDEFINED.

Otherwise, the allocation of encodings in this space is shown in Table A7-22. Other encodings in this space are UNDEFINED.

These instructions are `MRC` and `MCR` instructions for coprocessors 10 and 11.

Table A7-22 8-bit, 16-bit and 32-bit data transfer instructions

L	C	A	B	Instruction	See
0	0	000	-	Vector Move	*VMOV (between ARM core register and single-precision register)* on page A8-648
		111	-	Move to VFP Special Register from ARM core register	*VMSR* on page A8-660 *VMSR* on page B6-29 (System level view)
0	1	0xx	-	Vector Move	*VMOV (ARM core register to scalar)* on page A8-644
		1xx	0x	Vector Duplicate	*VDUP (ARM core register)* on page A8-594
1	0	000	-	Vector Move	*VMOV (between ARM core register and single-precision register)* on page A8-648
		111	-	Move to ARM core register from VFP Special Register	*VMRS* on page A8-658 *VMRS* on page B6-27 (System level view)
	1	xxx	-	Vector Move	*VMOV (scalar to ARM core register)* on page A8-646

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.8 ARM コアレジスタと拡張機能レジスタとの 8、16、32 ビット転送

Thumb エンコード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	T	1	1	1	0	A			L									1	0	1	C		B			1			

ARM エンコード

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	A			L									1	0	1	C		B			1			

Thumb エンコードで `T` == 1 のとき、または ARM エンコードで `cond` == 0b1111 のとき、命令は未定義です。

それ以外の場合、この空間におけるエンコードの割り当てを表 A7-22 に示します。この空間に含まれる他のエンコードは未定義です。

これらの命令は、コプロセッサ 10、11 の `MRC` および `MCR` 命令です。

表 A7-22 8、16、32 ビットのデータ転送命令

L	C	A	B	命令	参照先
0	0	000	–	ベクタ移動	*VMOV（ARM コアレジスタと単精度レジスタとの間の転送）* P. A8-648
		111	–	ARM コアレジスタから VFP 特殊レジスタへの移動	*VMSR* P. A8-660 *VMSR* P. B6-29（システムレベルのビュー）
0	1	0xx	–	ベクタ移動	*VMOV（ARM コアレジスタからスカラへのコピー）* P. A8-644
		1xx	0x	ベクタ複製	*VDUP（ARM コアレジスタ）* P. A8-594
1	0	000	–	ベクタ移動	*VMOV（ARM コアレジスタと単精度レジスタとの間の転送）* P. A8-648
		111	–	VFP 特殊レジスタから ARM コアレジスタへの移動	*VMRS* P. A8-658 *VMRS* P. B6-27（システムレベルのビュー）
	1	xxx	–	ベクタ移動	*VMOV（スカラから ARM コアレジスタへのコピー）* P. A8-646

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.9 64-bit transfers between ARM core and extension registers

Thumb encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	T	1	1	0	0	0	1	0										1	0	1	C	op							

ARM encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	0	0	1	0										1	0	1	C	op							

If `T` == 1 in the Thumb encoding or `cond` == 0b1111 in the ARM encoding, the instruction is UNDEFINED.

Otherwise, the allocation of encodings in this space is shown in Table A7-23. Other encodings in this space are UNDEFINED.

These instructions are `MRRC` and `MCRR` instructions for coprocessors 10 and 11.

Table A7-23 8-bit, 16-bit and 32-bit data transfer instructions

C	op	Instruction
0	00x1	*VMOV (between two ARM core registers and two single-precision registers)* on page A8-650
1	00x1	*VMOV (between two ARM core registers and a doubleword extension register)* on page A8-652

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A7.9 ARM コアレジスタと拡張機能レジスタとの 64 ビット転送

Thumb エンコード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	T	1	1	0	0	0	1	0										1	0	1	C	op							

ARM エンコード

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	0	0	1	0										1	0	1	C	op							

Thumb エンコードで `T` == 1 のとき、または ARM エンコードで `cond` == 0b1111 のとき、命令は未定義です。

それ以外の場合、この空間におけるエンコードの割り当てを表 A7-23 に示します。この空間に含まれる他のエンコードは未定義です。

これらの命令は、コプロセッサ 10、11 の `MRRC` および `MCRR` 命令です。

表 A7-23 8、16、32 ビットのデータ転送命令

C	op	命令
0	00x1	*VMOV（2 つの ARM コアレジスタと 2 つの単精度レジスタとの間の転送）* P. A8-650
1	00x1	*VMOV（2 つの ARM コアレジスタとダブルワード拡張機能レジスタとの間のコピー）* P. A8-652

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter A8
Instruction Details

This chapter describes each instruction. It contains the following sections:

- *Format of instruction descriptions* on page A8-2
- *Standard assembler syntax fields* on page A8-7
- *Conditional execution* on page A8-8
- *Shifts applied to a register* on page A8-10
- *Memory accesses* on page A8-13
- *Alphabetical list of instructions* on page A8-14.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 A8 章
命令の詳細

本章では、各命令について説明します。本章は次のセクションから構成されています。

- *命令の説明フォーマット* P. A8-2
- *標準アセンブラ構文のフィールド* P. A8-7
- *条件付き実行* P. A8-8
- *レジスタに適用されるシフト* P. A8-10
- *メモリアクセス* P. A8-13
- *命令のアルファベット順リスト* P. A8-14

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.1 Format of instruction descriptions

The instruction descriptions in *Alphabetical list of instructions* on page A8-14 normally use the following format:

- instruction section title
- introduction to the instruction
- instruction encoding(s) with architecture information
- assembler syntax
- pseudocode describing how the instruction operates
- exception information
- notes (where applicable).

Each of these items is described in more detail in the following subsections.

A few instruction descriptions describe alternative mnemonics for other instructions and use an abbreviated and modified version of this format.

A8.1.1 Instruction section title

The instruction section title gives the base mnemonic for the instructions described in the section. When one mnemonic has multiple forms described in separate instruction sections, this is followed by a short description of the form in parentheses. The most common use of this is to distinguish between forms of an instruction in which one of the operands is an immediate value and forms in which it is a register.

Parenthesized text is also used to document the former mnemonic in some cases where a mnemonic has been replaced entirely by another mnemonic in the new assembler syntax.

A8.1.2 Introduction to the instruction

The instruction section title is followed by text that briefly describes the main features of the instruction. This description is not necessarily complete and is not definitive. If there is any conflict between it and the more detailed information that follows, the latter takes priority.

A8.1.3 Instruction encodings

This is a list of one or more instruction encodings. Each instruction encoding is labelled as:

- T1, T2, T3 … for the first, second, third and any additional Thumb encodings
- A1, A2, A3 … for the first, second, third and any additional ARM encodings
- E1, E2, E3 … for the first, second, third and any additional ThumbEE encodings that are not also Thumb encodings.

Where Thumb and ARM encodings are very closely related, the two encodings are described together, for example as encoding T1 / A1.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.1 命令の説明フォーマット

命令のアルファベット順リスト P. A8-14 の命令の説明では通常、次のフォーマットを使用しています。

- 命令セクションのタイトル
- 命令の紹介
- 命令のエンコードとアーキテクチャ情報
- アセンブラ構文
- 命令の動作を説明する擬似コード
- 例外情報
- 注（該当する場合）

それぞれの項目について、以後のサブセクションで詳しい説明が行われます。

命令の説明によっては、他の命令の代替ニーモニックが紹介されたり、このフォーマットの省略バージョンや変更されたバージョンが使用されたりする場合があります。

A8.1.1 命令セクションのタイトル

命令セクションのタイトルでは、そのセクションで説明する命令の基本のニーモニックが示されます。ひとつのニーモニックが別の命令セクションとして記述される複数の形式を持つ場合、ニーモニックの後に括弧内でその形式の簡単な説明が示されます。この説明は主に、オペランドの 1 つがイミディエート値である形式と、レジスタを使用する形式を区別するために使用されます。

新しいアセンブラ構文でニーモニックが完全に別のものに置き換えられている場合、以前のニーモニックが括弧内に示されます。

A8.1.2 命令の紹介

命令セクションのタイトルの次に、その命令の主な機能に関する短い説明が続きます。この説明は、必ずしも完全ではなく、命令の明確な定義ではありません。この説明とその後に続く詳細情報が矛盾する場合は、後者が優先されます。

A8.1.3 命令のエンコード

これは 1 つ以上の命令のエンコードのリストです。命令のエンコードそれぞれに、次のようなラベルが付けられています。

- T1、T2、T3 … 第 1、第 2、第 3、および任意の追加 Thumb エンコード
- A1、A2、A3 … 第 1、第 2、第 3、および任意の追加 ARM エンコード
- E1、E2、E3 … Thumb エンコードではない、第 1、第 2、第 3、および任意の追加 ThumbEE エンコード

Thumb エンコードと ARM エンコードは非常に密接な関係にあります。この 2 つのエンコードは、例えばエンコード T1/A1 として一緒に説明します。

命令のエンコードの説明には、次の項目が含まれています。

- 命令の特定のエンコードを含むアーキテクチャバリアントに関する情報。この情報は、次のいずれかの方法で示されます。
  - 主要な命令セットアーキテクチャに含まれる命令のエンコードの場合、エンコードを含むアーキテクチャバリアントのリストとして示されます。これらのバリアントの要約については、*アーキテクチャのバージョン*、*プロファイル*、*バリアント* P. A1-4 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Each instruction encoding description consists of:

- Information about which architecture variants include the particular encoding of the instruction. This is presented in one of two ways:
  - — For instruction encodings that are in the main instruction set architecture, as a list of the architecture variants that include the encoding. See *Architecture versions, profiles, and variants* on page A1-4 for a summary of these variants.
  - — For instruction encodings that are in the architecture extensions, as a list of the architecture extensions that include the encoding. See *Architecture extensions* on page A1-6 for a summary of the architecture extensions and the architecture variants that they can extend.

  In architecture variant lists:
  - — ARMv7 means ARMv7-A and ARMv7-R profiles. The architecture variant information in this manual does not cover the ARMv7-M profile.
  - — * is used as a wildcard. For example, ARMv5T* means ARMv5T, ARMv5TE, and ARMv5TEJ.
- An assembly syntax that ensures that the assembler selects the encoding in preference to any other encoding. In some cases, multiple syntaxes are given. The correct one to use is sometimes indicated by annotations to the syntax, such as *Inside IT block* and *Outside IT block*. In other cases, the correct one to use can be determined by looking at the assembler syntax description and using it to determine which syntax corresponds to the instruction being disassembled.

  There is usually more than one syntax that ensures re-assembly to any particular encoding, and the exact set of syntaxes that do so usually depends on the register numbers, immediate constants and other operands to the instruction. For example, when assembling to the Thumb instruction set, the syntax `AND R0,R0,R8` ensures selection of a 32-bit encoding but `AND R0,R0,R1` selects a 16-bit encoding.

  The assembly syntax documented for the encoding is chosen to be the simplest one that ensures selection of that encoding for all operand combinations supported by that encoding. This often means that it includes elements that are only necessary for a small subset of operand combinations. For example, the assembler syntax documented for the 32-bit Thumb `AND` (register) encoding includes the `.W` qualifier to ensure that the 32-bit encoding is selected even for the small proportion of operand combinations for which the 16-bit encoding is also available.

  The assembly syntax given for an encoding is therefore a suitable one for a disassembler to disassemble that encoding to. However, disassemblers might wish to use simpler syntaxes when they are suitable for the operand combination, in order to produce more readable disassembled code.
- An encoding diagram, or a Thumb encoding diagram followed by an ARM encoding diagram when they are being described together. This is half-width for 16-bit Thumb encodings and full-width for 32-bit Thumb and ARM encodings. The 32-bit Thumb encodings use a double vertical line between the two halfwords of the instruction to distinguish them from ARM encodings and to act as a reminder that 32-bit Thumb instructions consist of two consecutive halfwords rather than a word.

  In particular, if instructions are stored using the standard little-endian instruction endianness, the encoding diagram for an ARM instruction at address A shows the bytes at addressees A+3, A+2, A+1, A from left to right, but the encoding diagram for a 32-bit Thumb instruction shows them in the order A+1, A for the first halfword, followed by A+3, A+2 for the second halfword.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- — アーキテクチャ拡張機能に含まれる命令のエンコードの場合、エンコードを含むアーキテクチャ拡張機能のリストとして示されます。アーキテクチャ拡張機能と、アーキテクチャ拡張機能によって拡張できるアーキテクチャバリアントの要約については、*アーキテクチャ拡張機能* P. A1-6 を参照して下さい。

  アーキテクチャバリアントのリストには、次の規則が適用されます。
  - — ARMv7 は、ARMv7-A プロファイルと ARMv7-R プロファイルを意味します。本書のアーキテクチャバリアント情報には、ARMv7-M プロファイルは含まれていません。
  - — * はワイルドカードです。例えば、ARMv5T* は ARMv5T、ARMv5TE、および ARMv5TEJ を意味します。
- アセンブラがそのエンコードを他のどのエンコードよりも優先的に選択することが保証されているアセンブラ構文。複数の構文が示される場合があります。使用すべき適切な構文が、*IT ブロック内部*や *IT ブロック外部*などの構文に付加された注釈によって示されている場合もあります。それ以外の場合では、アセンブラ構文の説明を読んで、逆アセンブルした命令がどの構文に対応するかを理解することによって、適切な構文を判定できます。

  通常は、何らかの特定のエンコードに再アセンブルできる構文が複数存在します。このような構文の正確なセットは、一般にレジスタ番号、イミディエート定数、および命令に対するその他のオペランドによって異なります。例えば、Thumb 命令セットにアセンブルする場合、`AND R0,R0,R8` という構文では 32 ビットエンコードが必ず選択されますが、`AND R0,R0,R1` という構文では 16 ビットエンコードが選択されます。

  エンコード用に記述されているアセンブラ構文は、そのエンコードでサポートされているすべてのオペランドの組み合わせの中で、最も単純なものが選択されています。このため多くの場合、そのアセンブラ構文には、オペランドの組み合わせのうちの一部に対してのみ必要な要素が含まれています。例えば、32 ビット Thumb の `AND`（レジスタ）エンコードについて記載されているアセンブラ構文では、一部のオペランドの組み合わせについては 16 ビットエンコードが使用可能ですが、そのような組み合わせについても、必ず 32 ビットエンコードが選択されるように、`.W` 修飾子が使用されています。

  したがって、あるエンコードに対して付加されたアセンブラ構文は、そのエンコードを逆アセンブルするための逆アセンブラでの使用にも適しています。ただし、逆アセンブラでは、逆アセンブルされたコードを読みやすくするため、特定のオペランドの組み合わせに適した、より単純な構文を使用してもかまいません。
- エンコード図、または Thumb エンコード図と ARM エンコード図（Thumb エンコード図と ARM エンコード図を一緒に説明する場合）。16 ビット Thumb エンコード用の半分の幅の図と、32 ビット Thumb エンコードおよび ARM エンコード用の完全な幅の図があります。32 ビット Thumb エンコードでは、ARM エンコードと区別するためと、32 ビット Thumb 命令がワードではなく 2 つの連続するハーフワードで構成されることを示すために、2 つのハーフワードの間に縦の二重線を使用しています。

  特に、標準的なリトルエンディアン命令のエンディアン形式を使用して命令がストアされる場合、アドレス A での ARM 命令のエンコード図には、左から右にアドレス A+3、A+2、A+1、A のバイトが示されます。これに対し、32 ビット Thumb 命令のエンコード図には、最初のハーフワードのバイトが A+1、A の順に示され、それに続いて 2 番目のハーフワードのバイトが A+3、A+2 の順に示されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- Encoding-specific pseudocode. This is pseudocode that translates the encoding-specific instruction fields into inputs to the encoding-independent pseudocode in the later *Operation* subsection, and that picks out any special cases in the encoding. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Appendix I *Pseudocode Definition*.

A8.1.4 Assembler syntax

The *Assembly syntax* subsection describes the standard UAL syntax for the instruction.

Each syntax description consists of the following elements:

- One or more syntax prototype lines written in a `typewriter` font, using the conventions described in *Assembler syntax prototype line conventions* on page A8-5. Each prototype line documents the mnemonic and (where appropriate) operand parts of a full line of assembler code. When there is more than one such line, each prototype line is annotated to indicate required results of the encoding-specific pseudocode. For each instruction encoding, this information can be used to determine whether any instructions matching that encoding are available when assembling that syntax, and if so, which ones.

- The line *where:* followed by descriptions of all of the variable or optional fields of the prototype syntax line.

  Some syntax fields are standardized across all or most instructions. *Standard assembler syntax fields* on page A8-7 describes these fields.

  By default, syntax fields that specify registers, such as `<Rd>`, `<Rn>`, or `<Rt>`, can be any of R0-R12 or LR in Thumb instructions, and any of R0-R12, SP or LR in ARM instructions. These require that the encoding-specific pseudocode set the corresponding integer variable (such as `d`, `n`, or `t`) to the corresponding register number (0-12 for R0-R12, 13 for SP, 14 for LR). This can normally be done by setting the corresponding bitfield in the instruction (named Rd, Rn, Rt…) to the binary encoding of that number. In the case of 16-bit Thumb encodings, this bitfield is normally of length 3 and so the encoding is only available when one of R0-R7 is specified in the assembler syntax. It is also common for such encodings to use a bitfield name such as Rdn. This indicates that the encoding is only available if `<Rd>` and `<Rn>` specify the same register, and that the register number of that register is encoded in the bitfield if they do.

  The description of a syntax field that specifies a register sometimes extends or restricts the permitted range of registers or documents other differences from the default rules for such fields. Typical extensions are to permit the use of the SP in Thumb instructions and to permit the use of the PC (using register number 15).

- Where appropriate, text that briefly describes changes from the pre-UAL ARM assembler syntax. Where present, this usually consists of an alternative pre-UAL form of the assembler mnemonic. The pre-UAL ARM assembler syntax does not conflict with UAL, and support for it is a recommended optional extension to UAL, to enable the assembly of pre-UAL ARM assembler source files.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- エンコード固有の擬似コード。これは、エンコード固有の命令フィールドを、後述の「*動作*」サブセクション内のエンコード独立の擬似コードへの入力に変換し、エンコードの特殊な条件を抽出する擬似コードです。使用されている擬似コードの詳細と、エンコード図、エンコード固有の擬似コード、およびエンコード独立の擬似コード間の関係については、付録 I *「擬似コードの定義」*を参照して下さい。

A8.1.4 アセンブラ構文

「*アセンブラ構文*」サブセクションには、その命令について標準的な UAL 構文が記載されています。

各構文の説明には、次の項目が含まれています。

- *アセンブラ構文プロトタイプ行の規則* P. A8- 5 で説明されている規則を使用して `typewriter` フォントで記載された 1 つ以上の構文プロトタイプ行。各プロトタイプ行には、一行全体のアセンブラコードに含まれる、ニーモニックとオペランドの部分（該当する場合）が記載されています。このような行が複数ある場合は、各プロトタイプ行に、エンコード固有の擬似コードに必要な結果を示す注釈が付けられます。命令エンコードのそれぞれについて、その構文をアセンブルする際にそのエンコードに適合する何らかの命令があるか、もしあるならどれかを、この情報を使用して決定できます。

- プロトタイプ構文行のすべての変数またはオプションフィールドの説明が続く「*各項目の説明については以下を参照して下さい*」行。

  一部の構文フィールドは、すべての命令またはほとんどの命令について標準化されています。このようなフィールドについては、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

  デフォルトで、レジスタを指定する構文フィールド（`<Rd>`、`<Rn>`、`<Rt>` など）は、Thumb 命令内で R0 ～ R12 または LR のいずれかに、ARM 命令内で R0 ～ R12、SP、または LR のいずれかに、それぞれ置き換えることができます。この場合は、エンコード固有の擬似コードで、整数変数（`d`、`n`、`t` など）を対応するレジスタ番号（R0 ～ R12 の場合は 0 ～ 12、SP の場合は 13、LR の場合は 14）に設定する必要があります。通常は、命令内の対応するビットフィールド（Rd、Rn、Rt などの名前が付けられています）を、その番号のバイナリエンコードに設定することによって実現されます。16 ビット Thumb エンコードの場合は、このビットフィールドの長さが通常は 3 であるため、アセンブラ構文で R0 ～ R7 のいずれかが指定されている場合にのみこのエンコードが使用できます。また、このようなエンコードでは、Rdn などのビットフィールド名が多く使用されます。これは、`<Rd>` と `<Rn>` が同じレジスタを指している場合にのみこのエンコードが使用でき、その場合は、そのレジスタの番号がビットフィールドにエンコードされることを意味しています。

  レジスタを指定する構文フィールドの説明では、レジスタの許容範囲が拡張または制限されている場合や、その他フィールドのデフォルトの規則とは相違点があることが記載されている場合があります。このような拡張の例として、Thumb 命令での SP の使用や PC の使用（レジスタ番号 15 を使用します）が許可される場合が挙げられます。

- UAL 以前の ARM アセンブラ構文からの変更に関する簡潔な説明（該当する場合）。この説明には通常、UAL 以前の代替形式のアセンブラニーモニックが含まれます。UAL 以前の ARM アセンブラ構文は UAL と競合しません。このアセンブラ構文をサポートすることは、UAL 以前の ARM アセンブラ ソースファイルのアセンブリを可能にするために、UAL に対するオプションの拡張機能として推奨されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

The pre-UAL Thumb assembler syntax is incompatible with UAL and is not documented in the instruction sections. For details see Appendix C *Legacy Instruction Mnemonics*.

Assembler syntax prototype line conventions

The following conventions are used in assembler syntax prototype lines and their subfields:

< >
: Any item bracketed by < and > is a short description of a type of value to be supplied by the user in that position. A longer description of the item is normally supplied by subsequent text. Such items often correspond to a similarly named field in an encoding diagram for an instruction. When the correspondence simply requires the binary encoding of an integer value or register number to be substituted into the instruction encoding, it is not described explicitly. For example, if the assembler syntax for an ARM instruction contains an item `<Rn>` and the instruction encoding diagram contains a 4-bit field named Rn, the number of the register specified in the assembler syntax is encoded in binary in the instruction field.

  If the correspondence between the assembler syntax item and the instruction encoding is more complex than simple binary encoding of an integer or register number, the item description indicates how it is encoded. This is often done by specifying a required output from the encoding-specific pseudocode, such as `add = TRUE`. The assembler must only use encodings that produce that output.

{ }
: Any item bracketed by { and } is optional. A description of the item and of how its presence or absence is encoded in the instruction is normally supplied by subsequent text.

  Many instructions have an optional destination register. Unless otherwise stated, if such a destination register is omitted, it is the same as the immediately following source register in the instruction syntax.

spaces
: Single spaces are used for clarity, to separate items. When a space is obligatory in the assembler syntax, two or more consecutive spaces are used.

+/-
: This indicates an optional + or - sign. If neither is coded, + is assumed.

All other characters must be encoded precisely as they appear in the assembler syntax. Apart from { and }, the special characters described above do not appear in the basic forms of assembler instructions documented in this manual. The { and } characters need to be encoded in a few places as part of a variable item. When this happens, the long description of the variable item indicates how they must be used.

A8.1.5 Pseudocode describing how the instruction operates

The *Operation* subsection contains encoding-independent pseudocode that describes the main operation of the instruction. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Appendix I *Pseudocode Definition*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

UAL 以前の Thumb アセンブラ構文は、UAL と互換性がないため、命令セクションでは使用されていません。詳細については、付録 C *「過去の命令のニーモニック」*を参照して下さい。

アセンブラ構文プロトタイプ行の規則

アセンブラ構文プロトタイプ行とそのサブフィールドには、次の規則が適用されます。

< >
: < と > で囲まれた項目は、ユーザがその位置に指定する必要のある値のタイプに関する短い説明です。通常、その項目に関するより詳しい説明が後に続きます。このような項目の多くは、命令のエンコード図で同様の名前が付けられたフィールドに対応しています。この対応で、命令のエンコードに代入する整数値またはレジスタ番号のバイナリエンコードだけが必要な場合、その説明は省略されます。例えば、ARM 命令のアセンブラ構文に `<Rn>` という項目が含まれており、命令のエンコード図に Rn という名前の 4 ビットフィールドが含まれている場合は、アセンブラ構文で指定されたレジスタの番号がバイナリで命令フィールドにエンコードされます。

  アセンブラ構文の項目と命令のエンコードの間の対応が、整数やレジスタ番号の単純なバイナリエンコードよりも複雑な場合は、その項目の説明でどのようにエンコードするかが示されています。多くの場合、エンコード固有の擬似コードからの必須の出力を `add = TRUE` などで指定することによって実現されます。アセンブラでは、この出力を生成するエンコード以外は使用しないようにする必要があります。

{ }
: { と } で囲まれた項目はオプションです。通常、この項目の説明と、この項目が存在する場合と存在しない場合に命令がどのようにエンコードされるかの説明が後に続きます。

  多くの命令で、オプションとしてデスティネーションレジスタが使用されます。特記されていない限り、デスティネーションレジスタが省略されている場合は、命令構文で直後にあるソースレジスタと同一のレジスタが使用されます。

スペース
: 項目を分離してわかりやすくするために単一のスペースが使用されます。アセンブラ構文でスペースが必須の場合は、連続する複数のスペースが使用されます。

+/-
: オプションの +/- 符号を示しています。コードにどちらも指定されていない場合、+ と見なされます。

他のすべての文字は、アセンブラ構文に記載されているのと正確に同じにエンコードする必要があります。本書内のアセンブラ命令の基本形式には、{ および } を除いて、上に説明されている特殊文字は記載されていません。{ および } 文字は、一部の場所で、変数項目の一部としてエンコードする必要があります。これらの文字が使用されている部分では、変数項目の説明でそれらの使用方法が示されます。

A8.1.5 命令の動作を説明する擬似コード

「*動作*」サブセクションには、その命令の主要な動作を説明する、エンコード独立の擬似コードが含まれています。使用されている擬似コードの詳細と、エンコード図、エンコード固有の擬似コード、およびエンコード独立の擬似コード間の関係については、付録 I *「擬似コードの定義」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.1.6 Exception information

The *Exceptions* subsection contains a list of the exceptional conditions that can be caused by execution of the instruction.

Processor exceptions are listed as follows:

- Resets and interrupts (both IRQs and FIQs) are not listed. They can occur before or after the execution of any instruction, and in some cases during the execution of an instruction, but they are not in general caused by the instruction concerned.
- Prefetch Abort exceptions are normally caused by a memory abort when an instruction is fetched, followed by an attempt to execute that instruction. This can happen for any instruction, but is caused by the aborted attempt to fetch the instruction rather than by the instruction itself, and so is not listed. A special case is the `BKPT` instruction, that is defined as causing a Prefetch Abort exception in some circumstances.
- Data Abort exceptions are listed for all instructions that perform data memory accesses.
- Undefined Instruction exceptions are listed when they are part of the effects of a defined instruction. For example, all coprocessor instructions are defined to produce the Undefined Instruction exception if not accepted by their coprocessor. Undefined Instruction exceptions caused by the execution of an `UNDEFINED` instruction are not listed, even when the `UNDEFINED` instruction is a special case of one or more of the encodings of the instruction. Such special cases are instead indicated in the encoding-specific pseudocode for the encoding.
- Supervisor Call and Secure Monitor Call exceptions are listed for the `SVC` and `SMC` instructions respectively. Supervisor Call exceptions and the `SVC` instruction were previously called Software Interrupt exceptions and the `SWI` instruction. Secure Monitor Call exceptions and the `SMC` instruction were previously called Secure Monitor interrupts and the `SMI` instruction.

Floating-point exceptions are listed for instructions that can produce them. *Floating-point exceptions* on page A2-42 describes these exceptions. They do not normally result in processor exceptions.

A8.1.7 Notes

Where appropriate, other notes about the instruction appear under additional subheadings.

Note

Information that was documented in notes in previous versions of the ARM Architecture Reference Manual and its supplements has often been moved elsewhere. For example, operand restrictions on the values of bitfields in an instruction encoding are now normally documented in the encoding-specific pseudocode for that encoding.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.1.6 例外情報

「*例外*」サブセクションには、命令の実行によって発生する可能性のある例外状態のリストが含まれています。

プロセッサ例外は、次の規則に従って記載されています。

- リセットと割り込み（IRQ と FIQ の両方）は記載されていません。これらは、任意の命令の実行前または実行後、および特定の命令の実行中に発生する可能性がありますが、命令によって発生することは通常はありません。
- プリフェッチアボート例外は通常、命令をフェッチした後でその命令の実行を試みる際のメモリアボートによって発生します。これはすべての命令で発生する可能性があります。ただし、命令自体ではなく命令のフェッチ試行がアボートされることによって発生するため、記載されていません。特殊なケースとして、`BKPT` 命令は、状況によってプリフェッチアボート例外を引き起こす命令と定義されています。
- データアボート例外は、データメモリ アクセスを実行するすべての命令に対して記載されています。
- 未定義命令例外は、定義されている命令の効果の一部である場合に記載されています。例えば、コプロセッサ命令はすべて、そのコプロセッサによって受け付けられない場合に未定義命令例外を生成するものとして定義されています。未定義命令の実行によって発生する未定義命令例外は、未定義命令が命令の 1 つ以上のエンコードの特殊なケースである場合でも記載されていません。その代わりに、このような特殊なケースはエンコード固有の擬似コードに示されます。
- スーパバイザコール例外とセキュア モニタコール例外はそれぞれ、`SVC` 命令と `SMC` 命令に対して記載されています。以前は、スーパバイザコール例外はソフトウェア割り込み例外、`SVC` 命令は `SWI` 命令とそれぞれ呼ばれていました。同様に、セキュア モニタコール例外はセキュアモニタ割り込み、`SMC` 命令は `SMI` 命令とそれぞれ呼ばれていました。

浮動小数点例外は、浮動小数点を生成できる命令に対して記載されています。これらの例外について、*浮動小数点例外* P. A2- 42 を参照して下さい。通常は、これらの例外が原因でプロセッサ例外が発生することはありません。

A8.1.7 注

必要に応じて、小見出しの下に命令に関する注記が記載されています。

注

以前のバージョンの ARM アーキテクチャ リファレンスマニュアルとその付録の注に記載されている情報は、多くの場合、別の場所に移されています。例えば、命令のエンコードでのビットフィールドの値に対するオペランドの制約は通常、そのエンコードのエンコード固有の擬似コードに示されるようになりました。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.2 Standard assembler syntax fields

The following assembler syntax fields are standard across all or most instructions:

<c> Is an optional field. It specifies the condition under which the instruction is executed. See *Conditional execution* on page A8-8 for the range of available conditions and their encoding. If <c> is omitted, it defaults to *always* (AL).

<q> Specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N Meaning narrow, specifies that the assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.
- .W Meaning wide, specifies that the assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither .W nor .N is specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding. In a few cases, more than one encoding of the same length can be available for an instruction. The rules for selecting between such encodings are instruction-specific and are part of the instruction description.

Note

When assembling to the ARM instruction set, the .N qualifier produces an assembler error and the .W qualifier has no effect.

Although the instruction descriptions throughout this manual show the <c> and <q> fields without { } around them, these fields are optional as described in this section.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.2 標準アセンブラ構文のフィールド

次のアセンブラ構文フィールドは、ほぼすべての命令で共通しています。

<c> オプションフィールド。この命令が実行される条件が示されます。使用可能な条件の範囲とそのエンコードについては、*条件付き実行* P. A8-8 を参照して下さい。<c> が省略されている場合、デフォルトで *常時* (AL) に設定されます。

<q> 命令に関するオプションのアセンブラ修飾子を指定します。次の修飾子が定義されています。

- .N Narrow（狭い）を意味し、その命令に対してアセンブラが 16 ビットエンコードを選択する必要があることを指定します。これが不可能な場合は、アセンブラエラーが発生します。
- .W Wide（広い）を意味し、その命令に対してアセンブラが 32 ビットエンコードを選択する必要があることを示します。これが不可能な場合は、アセンブラエラーが発生します。

.W と .N のどちらも指定されていない場合、アセンブラが 16 ビットエンコードと 32 ビットエンコードのいずれかを選択します。両方が使用可能な場合、アセンブラは 16 ビットエンコードを選択する必要があります。1 つの命令に対して同じ長さの複数のエンコードが使用可能な場合があります。どのエンコードを選択するかの規則は命令によって異なり、命令の説明で示されています。

注

ARM 命令セットにアセンブルする場合、.N 修飾子はアセンブラエラーを生成しますが、.W 修飾子は何も影響しません。

本書では、命令の説明で <c> フィールドと <q> フィールドを { } で囲まないで示していますが、このセクションで説明したように、これらのフィールドはオプションです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.3 Conditional execution

Most ARM instructions, and most Thumb instructions from ARMv6T2 onwards, can be executed conditionally, based on the values of the APSR condition flags. Before ARMv6T2, the only conditional Thumb instruction was the 16-bit conditional branch instruction. Table A8-1 lists the available conditions.

In Thumb instructions, the condition (if it is not AL) is normally encoded in a preceding IT instruction. For details see *Conditional instructions* on page A4-4 and *IT* on page A8-104. Some conditional branch instructions do not require a preceding IT instruction, and include a condition code in their encoding.

In ARM instructions, bits [31:28] of the instruction contain the condition, or contain 1111 for some ARM instructions that can only be executed unconditionally.

Table A8-1 Condition codes

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) [a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS [b]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC [c]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL) [d]	Always (unconditional)	Always (unconditional)	Any

a. Unordered means at least one NaN operand.
b. HS (unsigned higher or same) is a synonym for CS.
c. LO (unsigned lower) is a synonym for CC.
d. AL is an optional mnemonic extension for always, except in IT instructions. For details see *IT* on page A8-104.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.3 条件付き実行

ほとんどの ARM 命令と、ARMv6T2 およびそれ以降の Thumb 命令の大半は、APSR 条件フラグの値に基づいて、条件付きで実行することができます。ARMv6T2 以前は、条件付きの Thumb 命令は 16 ビット条件付き分岐命令のみでした。使用可能な条件のリストを表 A8-1 に示します。

Thumb 命令では、条件（AL 以外の場合）は一般に、先行する IT 命令にエンコードされます。詳細については、*条件付き命令* P. A4-4 および *IT* P. A8-104 を参照して下さい。条件付き分岐命令によっては、エンコードに条件コードが含まれており、前の IT 命令が必要ないものもあります。

ARM 命令では、命令のビット [31:28] に条件が含まれます。ただし、無条件でのみ実行できる一部の ARM 命令では、1111 が含まれます。

表 A8-1 条件コード

cond	ニーモニック拡張	意味（整数）	意味（浮動小数点）[a]	条件フラグ
0000	EQ	等しい	等しい	Z == 1
0001	NE	等しくない	等しくない、または規則なし	Z == 0
0010	CS [b]	キャリーセット	≧、または規則なし	C == 1
0011	CC [c]	キャリークリア	＜	C == 0
0100	MI	マイナス / 負	＜	N == 1
0101	PL	プラス、正、またはゼロ	≧、または規則なし	N == 0
0110	VS	オーバフロー	規則なし	V == 1
0111	VC	オーバフローなし	規則あり	V == 0
1000	HI	符号なし＞	＞、または規則なし	C＝1かつZ＝0
1001	LS	符号なし≦	≦	C＝0またはZ＝1
1010	GE	符号付き≧	≧	N == V
1011	LT	符号付き＜	＜、または規則なし	N != V
1100	GT	符号付き＞	＞	Z＝0かつN＝V
1101	LE	符号付き≦	≦、または規則なし	Z＝1またはN !=V
1110	なし（AL）[d]	常時（無条件）	常時（無条件）	任意

a. 規則なしは、少なくとも 1 つは NaN オペランドであることを意味します。
b. HS（符号なし≧）は、CS と同義語です。
c. LO（符号なし＜）は、CC と同義語です。
d. AL は、IT 命令を除いて、常時を意味するオプションのニーモニック拡張です。詳細については、*IT* P. A8-104 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.3.1 Pseudocode details of conditional execution

The `CurrentCond()` pseudocode function has prototype:

```
bits(4) CurrentCond()
```

and returns a 4-bit condition specifier as follows:

- For ARM instructions, it returns bits[31:28] of the instruction.
- For the T1 and T3 encodings of the Branch instruction (see *B* on page A8-44), it returns the 4-bit 'cond' field of the encoding.
- For all other Thumb and ThumbEE instructions, it returns ITSTATE.IT<7:4>. See *ITSTATE* on page A2-17.

The `ConditionPassed()` function uses this condition specifier and the APSR condition flags to determine whether the instruction must be executed:

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    cond = CurrentCond();

    // Evaluate base condition.
    case cond<3:1> of
        when '000'  result = (APSR.Z == '1');                        // EQ or NE
        when '001'  result = (APSR.C == '1');                        // CS or CC
        when '010'  result = (APSR.N == '1');                        // MI or PL
        when '011'  result = (APSR.V == '1');                        // VS or VC
        when '100'  result = (APSR.C == '1') && (APSR.Z == '0');     // HI or LS
        when '101'  result = (APSR.N == APSR.V);                     // GE or LT
        when '110'  result = (APSR.N == APSR.V) && (APSR.Z == '0');  // GT or LE
        when '111'  result = TRUE;                                   // AL

    // Condition bits '111x' indicate the instruction is always executed. Otherwise,
    // invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.3.1 条件付き実行の擬似コードの詳細

`CurrentCond()` 擬似コード関数には次のプロトタイプがあります。

```
bits(4) CurrentCond()
```

このプロトタイプは、次の 4 ビットの条件指定子を返します。

- ARM 命令では、命令のビット [31:28] を返します。
- 分岐命令（*B* P. A8- 44 参照）の T1 および T3 エンコードでは、4 ビットの cond フィールドを返します。
- 他のすべての Thumb 命令と ThumbEE 命令では、ITSTATE.IT〈7:4〉を返します。詳細については、*ITSTATE* P. A2- 17 を参照して下さい。

`ConditionPassed()` 関数は、この条件指定子と APSR 条件フラグを使用して、命令を実行すべきかどうかを判断します。

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    cond = CurrentCond();

    // Evaluate base condition.
    case cond<3:1> of
        when '000'  result = (APSR.Z == '1');                        // EQ or NE
        when '001'  result = (APSR.C == '1');                        // CS or CC
        when '010'  result = (APSR.N == '1');                        // MI or PL
        when '011'  result = (APSR.V == '1');                        // VS or VC
        when '100'  result = (APSR.C == '1') && (APSR.Z == '0');     // HI or LS
        when '101'  result = (APSR.N == APSR.V);                     // GE or LT
        when '110'  result = (APSR.N == APSR.V) && (APSR.Z == '0');  // GT or LE
        when '111'  result = TRUE;                                   // AL

    // Condition bits ?111x? indicate the instruction is always executed.Otherwise,
    // invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.4 Shifts applied to a register

ARM register offset load/store word and unsigned byte instructions can apply a wide range of different constant shifts to the offset register. Both Thumb and ARM data-processing instructions can apply the same range of different constant shifts to the second operand register. For details see *Constant shifts*.

ARM data-processing instructions can apply a register-controlled shift to the second operand register.

A8.4.1 Constant shifts

These are the same in Thumb and ARM instructions, except that the input bits come from different positions.

`<shift>` is an optional shift to be applied to `<Rm>`. It can be any one of:

(omitted) No shift.

`LSL #<n>` Logical shift left `<n>` bits. 1 <= `<n>` <= 31.

`LSR #<n>` Logical shift right `<n>` bits. 1 <= `<n>` <= 32.

`ASR #<n>` Arithmetic shift right `<n>` bits. 1 <= `<n>` <= 32.

`ROR #<n>` Rotate right `<n>` bits. 1 <= `<n>` <= 31.

`RRX` Rotate right one bit, with extend. Bit [0] is written to `shifter_carry_out`, bits [31:1] are shifted right one bit, and the Carry Flag is shifted into bit [31].

Note

Assemblers can permit the use of some or all of `ASR #0`, `LSL #0`, `LSR #0`, and `ROR #0` to specify that no shift is to be performed. This is not standard UAL, and the encoding selected for Thumb instructions might vary between UAL assemblers if it is used. To ensure disassembled code assembles to the original instructions, disassemblers must omit the shift specifier when the instruction specifies no shift.

Similarly, assemblers can permit the use of `#0` in the immediate forms of `ASR`, `LSL`, `LSR`, and `ROR` instructions to specify that no shift is to be performed, that is, that a `MOV` (register) instruction is wanted. Again, this is not standard UAL, and the encoding selected for Thumb instructions might vary between UAL assemblers if it is used. To ensure disassembled code assembles to the original instructions, disassemblers must use the `MOV` (register) syntax when the instruction specifies no shift.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.4 レジスタに適用されるシフト

ARM レジスタオフセットを使用する、ワードおよび符号なしバイトのロード / ストア命令では、各種の定数シフトをオフセットレジスタに適用することができます。Thumb および ARM のデータ処理命令では、同じ範囲の各種の定数シフトを 2 番目のオペランドレジスタに適用することができます。詳細については、*定数シフト*を参照して下さい。

ARM データ処理命令は、レジスタ制御シフトを 2 番目のオペランドレジスタに適用することができます。

A8.4.1 定数シフト

定数シフトは、入力ビットが異なる位置からくること以外は、Thumb 命令と ARM 命令とで同じです。

`<shift>` は、`<Rm>` に適用されるオプションのシフトです。次のいずれかのシフトを使用できます。

(省略) シフトなし

`LSL #<n>` `<n>` ビットの論理左シフト 1 (<= `<n>` <= 31)

`LSR #<n>` `<n>` ビットの論理右シフト 1 (<= `<n>` <= 32)

`ASR #<n>` `<n>` ビットの算術右シフト 1 (<= `<n>` <= 32)

`ROR #<n>` `<n>` ビットの右ローテート 1 (<= `<n>` <= 31)

`RRX` 拡張付き 1 ビット右ローテート。ビット [0] を `shifter_carry_out` に書き込み、ビット [31:1] を右に 1 ビットシフトし、キャリーフラグをビット [31] にシフトします。

注

アセンブラは、`ASR #0`、`LSL #0`、`LSR #0`、`ROR #0` の一部またはすべてを使用して、シフトを実行しないように指定することを許可できます。これは標準的な UAL ではありません。また、使用した場合に、Thumb 命令に対して選択されるエンコードが UAL アセンブラによって異なることがあります。逆アセンブルされたコードが元の命令にアセンブルされるように、逆アセンブラは、命令でシフトが指定されていない場合にシフト指定子を省略する必要があります。

同様に、アセンブラは、`#0` を `ASR`、`LSL`、`LSR`、`ROR` の各命令のイミディエート形式で使用して、シフトを実行しないように（つまり、`MOV`（レジスタ）命令が必要であることを）指定することを許可できます。これも標準的な UAL ではありません。また、使用した場合に、Thumb 命令に対して選択されるエンコードが UAL アセンブラによって異なることがあります。逆アセンブルされたコードが元の命令にアセンブルされるように、逆アセンブラは、命令でシフトが指定されていない場合に `MOV`（レジスタ）構文を使用する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Encoding

The assembler encodes <shift> into two type bits and five immediate bits, as follows:

(omitted) type = 0b00, immediate = 0.

LSL #<n> type = 0b00, immediate = <n>.

LSR #<n> type = 0b01.
If <n> < 32, immediate = <n>.
If <n> == 32, immediate = 0.

ASR #<n> type = 0b10.
If <n> < 32, immediate = <n>.
If <n> == 32, immediate = 0.

ROR #<n> type = 0b11, immediate = <n>.

RRX type = 0b11, immediate = 0.

A8.4.2 Register controlled shifts

These are only available in ARM instructions.

<type> is the type of shift to apply to the value read from <Rm>. It must be one of:

ASR Arithmetic shift right, encoded as type = 0b10
LSL Logical shift left, encoded as type = 0b00
LSR Logical shift right, encoded as type = 0b01
ROR Rotate right, encoded as type = 0b11.

The bottom byte of <Rs> contains the shift amount.

A8.4.3 Pseudocode details of instruction-specified shifts and rotates

```
enumeration SRType (SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX);

// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

    case type of
        when '00'
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

エンコード

アセンブラは、次に示すように <shift> を 2 つの type ビットと、5 つの immediate ビットにエンコードします。

(省略) type = 0b00、immediate = 0

LSL #<n> type = 0b00、immediate = <n>

LSR #<n> type = 0b01
<n> < 32 の場合、immediate = <n>
<n> == 32 の場合、immediate = 0

ASR #<n> type = 0b10
<n> < 32 の場合、immediate = <n>
<n> == 32 の場合、immediate = 0

ROR #<n> type = 0b11、immediate = <n>

RRX type = 0b11、immediate = 0

A8.4.2 レジスタ制御シフト

このシフトは、ARM 命令でのみ使用できます。

<type> は <Rm> から読み出された値に適用されるシフトのタイプです。次のいずれかを使用できます。

ASR 算術右シフト。type = 0b10 としてエンコードされます。
LSL 論理左シフト。type = 0b00 としてエンコードされます。
LSR 論理右シフト。type = 0b01 としてエンコードされます。
ROR 右ローテート。type = 0b11 としてエンコードされます。

<Rs> の下位バイトには、シフト量が含まれています。

A8.4.3 命令により指定されるシフトとローテートの擬似コードの詳細


```
enumeration SRType (SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX);

// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

    case type of
        when '00'
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
            if imm5 == '00000' then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);

// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) type)
    case type of
        when '00'  shift_t = SRType_LSL;
        when '01'  shift_t = SRType_LSR;
        when '10'  shift_t = SRType_ASR;
        when '11'  shift_t = SRType_ROR;
    return shift_t;

// Shift()
// =======

bits(N) Shift(bits(N) value, SRType type, integer amount, bit carry_in)
    (result, -) = Shift_C(value, type, amount, carry_in);
    return result;

// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳


```
            when '10'
                shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
            when '11'
                if imm5 == '00000' then
                    shift_t = SRType_RRX;  shift_n = 1;
                else
                    shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);

// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) type)
    case type of
        when '00'  shift_t = SRType_LSL;
        when '01'  shift_t = SRType_LSR;
        when '10'  shift_t = SRType_ASR;
        when '11'  shift_t = SRType_ROR;
    return shift_t;

// Shift()
// =======

bits(N) Shift(bits(N) value, SRType type, integer amount, bit carry_in)
    (result, -) = Shift_C(value, type, amount, carry_in);
    return result;

// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.5 Memory accesses

Commonly, the following addressing modes are permitted for memory access instructions:

Offset addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The value of the base register is unchanged.

The assembly language syntax for this mode is:

```
[<Rn>,<offset>]
```

Pre-indexed addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register.

The assembly language syntax for this mode is:

```
[<Rn>,<offset>]!
```

Post-indexed addressing

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register

The assembly language syntax for this mode is:

```
[<Rn>],<offset>
```

In each case, `<Rn>` is the base register. `<offset>` can be:

- an immediate constant, such as `<imm8>` or `<imm12>`
- an index register, `<Rm>`
- a shifted index register, such as `<Rm>, LSL #<shift>`.

For information about unaligned access, endianness, and exclusive access, see:

- *Alignment support* on page A3-4
- *Endian support* on page A3-7
- *Synchronization and semaphores* on page A3-12.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.5 メモリアクセス

一般に、次のアドレシングモードをメモリアクセス命令で使用することができます。

オフセットアドレシング

ベースレジスタから取得されたアドレスにオフセット値が適用されます。その結果が、メモリアクセスのアドレスとして使用されます。ベースレジスタの値は変更されません。

このモードのアセンブリ言語構文は次のとおりです。

```
[<Rn>,<offset>]
```

プリインデクス アドレシング

ベースレジスタから取得されたアドレスにオフセット値が適用されます。その結果がメモリアクセスのアドレスとして使用され、ベースレジスタに書き戻されます。

このモードのアセンブリ言語構文は次のとおりです。

```
[<Rn>,<offset>]!
```

ポストインデクス アドレシング

ベースレジスタから取得されたアドレスが、そのままメモリアクセス用のアドレスとして使用されます。オフセット値がアドレスに適用され、結果がベースレジスタに書き戻されます。

このモードのアセンブリ言語構文は次のとおりです。

```
[<Rn>],<offset>
```

それぞれの場合について、`<Rn>` はベースレジスタを表します。`<offset>` には、次のものを使用できます。

- `<imm8>` や `<imm12>` などのイミディエート定数
- インデクスレジスタ `<Rm>`
- `<Rm>, LSL #<shift>` などのシフトしたインデクスレジスタ

アンアラインドアクセス、エンディアン形式、排他アクセスの詳細については、次の項を参照して下さい。

- *アライメントのサポート* P. A3-4
- *エンディアンのサポート* P. A3-7
- *同期化とセマフォ* P. A3-12

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6 Alphabetical list of instructions

Every instruction is listed in this section. For details of the format used see *Format of instruction descriptions* on page A8-2.

A8.6.1 ADC (immediate)

Add with Carry (immediate) adds an immediate value and the carry flag value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv6T2, ARMv7

```
ADC{S}<c> <Rd>,<Rn>,#<const>
```

15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	0	1 0 1 0	S	Rn	0	imm3	Rd	imm8

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADC{S}<c> <Rd>,<Rn>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 1 0 1	S	Rn	Rd	imm12

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6 命令のアルファベット順リスト

このセクションには、すべての命令が記載されています。使用されているフォーマットの詳細については、*命令の説明フォーマット* P. A8-2 を参照して下さい。

A8.6.1 ADC（イミディエート）

ADC（キャリー付き加算（イミディエート））は、イミディエート値とキャリーフラグの値をレジスタの値に加算し、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

```
ADC{S}<c> <Rd>,<Rn>,#<const>
```

15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	0	1 0 1 0	S	Rn	0	imm3	Rd	imm8

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ADC{S}<c> <Rd>,<Rn>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 1 0 1	S	Rn	Rd	imm12

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ADC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<const>	The immediate value to be added to the value obtained from <Rn>. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

The pre-UAL syntax ADC<c>S is equivalent to ADCS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ADC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<const>	<Rn> から取得された値に加算されるイミディエート値。値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

UAL 以前の構文の ADC<c>S は、ADCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.2 ADC (register)

Add with Carry (register) adds a register value, the carry flag value, and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ADCS <Rdn>,<Rm>                    Outside IT block.
ADC<c> <Rdn>,<Rm>                  Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
ADC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	1	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADC{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	1	0	1	S	Rn				Rd				imm5					type		0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.2 ADC（レジスタ）

ADC（キャリー付き加算（レジスタ））は、レジスタの値、キャリーフラグの値、およびオプションでシフトされるレジスタの値を加算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ADCS <Rdn>,<Rm>                    IT ブロック外部
ADC<c> <Rdn>,<Rm>                  IT ブロック内部
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
ADC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	1	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ADC{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	1	0	1	S	Rn				Rd				imm5					type		0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ADC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The optionally shifted second operand register.

<shift> — The shift to apply to the value read from <Rm>. If present, encoding T1 is not permitted. If absent, no shift is applied and any encoding is permitted. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

In Thumb assembly:

- outside an IT block, if ADCS <Rd>,<Rn>,<Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ADCS <Rd>,<Rn> had been written.
- inside an IT block, if ADC<c> <Rd>,<Rn>,<Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ADC<c> <Rd>,<Rn> had been written.

To prevent either of these happening, use the .W qualifier.

The pre-UAL syntax ADC<c>S is equivalent to ADCS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ADC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — オプションとしてシフトされる 2 番目のオペランドレジスタ

<shift> — <Rm> から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。シフトとシフトのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

Thumb アセンブラでは、次の規則が適用されます。

- IT ブロック外部で、ADCS <Rd>,<Rn>,<Rd> の <Rd> と <Rn> の両方が R0 ～ R7 の範囲に含まれる場合、ADCS <Rd>,<Rn> と記述されたものとして、エンコード T1 を使用してアセンブルされます。
- IT ブロック内部で、ADC<c> <Rd>,<Rn>,<Rd> の <Rd> と <Rn> の両方が R0 ～ R7 の範囲に含まれる場合、ADC<c> <Rd>,<Rn> と記述されたものとして、エンコード T1 を使用してアセンブルされます。

これらの事態を避けるには、.W 修飾子を使用します。

UAL 以前の構文の ADC<c>S は、ADCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.3 ADC (register-shifted register)

Add with Carry (register-shifted register) adds a register value, the carry flag value, and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADC{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 1 0 1	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.3 ADC（レジスタよりシフトしたレジスタ）

ADC（キャリー付き加算（レジスタによりシフトしたレジスタ））は、レジスタの値、キャリーフラグの値、およびレジスタによりシフトしたレジスタの値を加算します。結果をデスティネーションレジスタに書き込み、オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ADC{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 1 0 1	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ADC{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The register that is shifted and used as the second operand.

<type> — The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR — Arithmetic shift right, encoded as type = 0b10
- LSL — Logical shift left, encoded as type = 0b00
- LSR — Logical shift right, encoded as type = 0b01
- ROR — Rotate right, encoded as type = 0b11.

<Rs> — The register whose bottom byte contains the amount to shift by.

The pre-UAL syntax ADC<c>S is equivalent to ADCS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ADC{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> — <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR — 算術右シフト。type = 0b10 としてエンコードされます。
- LSL — 論理左シフト。type = 0b00 としてエンコードされます。
- LSR — 論理右シフト。type = 0b01 としてエンコードされます。
- ROR — 右ローテート。type = 0b11 としてエンコードされます。

<Rs> — 最下位バイトにシフトのビット数を含むレジスタ

UAL 以前の構文の ADC<c>S は、ADCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.4 ADD (immediate, Thumb)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ADDS <Rd>,<Rn>,#<imm3>                    Outside IT block.
ADD<c> <Rd>,<Rn>,#<imm3>                  Inside IT block.
```

15 14 13	12 11	10	9	8 7 6	5 4 3	2 1 0
0 0 0	1 1	1	0	imm3	Rn	Rd

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ADDS <Rdn>,#<imm8>                        Outside IT block.
ADD<c> <Rdn>,#<imm8>                      Inside IT block.
```

15 14 13	12 11	10 9 8	7 6 5 4 3 2 1 0
0 0 1	1 0	Rdn	imm8

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Encoding T3 ARMv6T2, ARMv7

```
ADD{S}<c>.W <Rd>,<Rn>,#<const>
```

15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	0	1 0 0 0	S	Rn	0	imm3 Rd	imm8

```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || n == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv6T2, ARMv7

```
ADDW<c> <Rd>,<Rn>,#<imm12>
```

15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	1	0 0 0 0	0	Rn	0	imm3 Rd	imm8

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if BadReg(d) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.4 ADD（イミディエート、Thumb）

ADD（加算（イミディエート、Thumb））は、イミディエート値をレジスタの値に加算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ADDS <Rd>,<Rn>,#<imm3>                    IT ブロック外部
ADD<c> <Rd>,<Rn>,#<imm3>                  IT ブロック内部
```

15 14 13	12 11	10	9	8 7 6	5 4 3	2 1 0
0 0 0	1 1	1	0	imm3	Rn	Rd

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ADDS <Rdn>,#<imm8>                        IT ブロック外部
ADD<c> <Rdn>,#<imm8>                      IT ブロック内部
```

15 14 13	12 11	10 9 8	7 6 5 4 3 2 1 0
0 0 1	1 0	Rdn	imm8

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

エンコード T3 ARMv6T2、ARMv7

```
ADD{S}<c>.W <Rd>,<Rn>,#<const>
```

15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	0	1 0 0 0	S	Rn	0	imm3 Rd	imm8

```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || n == 15 then UNPREDICTABLE;
```

エンコード T4 ARMv6T2、ARMv7

```
ADDW<c> <Rd>,<Rn>,#<imm12>
```

15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	1	0 0 0 0	0	Rn	0	imm3 Rd	imm8

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if BadReg(d) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`ADD{S}<c><q> {<Rd>,} <Rn>, #<const>`	All encodings permitted
`ADDW<c><q> {<Rd>,} <Rn>, #<const>`	Only encoding T4 permitted

where:

`S` If `S` is present, the instruction updates the flags. Otherwise, the flags are not updated.

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rn>` The first operand register. If `<Rn>` is SP, see *ADD (SP plus immediate)* on page A8-28. If `<Rn>` is PC, see *ADR* on page A8-32.

`<const>` The immediate value to be added to the value obtained from `<Rn>`. The range of values is 0-7 for encoding T1, 0-255 for encoding T2 and 0-4095 for encoding T4. See *Modified immediate constants in Thumb instructions* on page A6-17 for the range of values for encoding T3.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the `ADDW` syntax). Encoding T1 is preferred to encoding T2 if `<Rd>` is specified and encoding T2 is preferred to encoding T1 if `<Rd>` is omitted.

The pre-UAL syntax `ADD<c>S` is equivalent to `ADDS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`ADD{S}<c><q> {<Rd>,} <Rn>, #<const>`	すべてのエンコードが許可される
`ADDW<c><q> {<Rd>,} <Rn>, #<const>`	エンコード T4 のみが許可される

各項目の説明については以下を参照して下さい。

`S` `S` を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rn>` 最初のオペランドレジスタ。`<Rn>` が SP の場合は、*ADD（SP + イミディエート）* P. A8- 28 を参照して下さい。`<Rn>` が PC の場合は、*ADR* P. A8- 32 を参照して下さい。

`<const>` `<Rn>` から取得された値に加算されるイミディエート値。値の範囲は、エンコード T1 では 0 ～ 7、エンコード T2 では 0 ～ 255、エンコード T4 では 0 ～ 4095 です。エンコード T3 での値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6- 17 を参照して下さい。

1 つの命令で同じ長さの複数のエンコードが使用可能な場合は、T4 エンコードよりも T3 エンコードの方が適しています（T4 エンコードが必要な場合は、`ADDW` 構文を使用して下さい）。`<Rd>` を指定する場合は、T2 エンコードよりも T1 エンコードの方が適しています。`<Rd>` を省略する場合は、T1 エンコードよりも T2 エンコードの方が適しています。

UAL 以前の構文の `ADD<c>S` は、`ADDS<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.5 ADD (immediate, ARM)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADD{S}<c> <Rd>,<Rn>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 1 0 0	S	Rn	Rd	imm12

```
if Rn == '1111' && S == '0' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.5 ADD（イミディエート、ARM）

ADD（加算（イミディエート、ARM））は、イミディエート値をレジスタの値に加算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ADD{S}<c> <Rd>,<Rn>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 1 0 0	S	Rn	Rd	imm12

```
if Rn == '1111' && S == '0' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ADD{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register. If the SP is specified for <Rn>, see *ADD (SP plus immediate)* on page A8-28. If the PC is specified for <Rn>, see *ADR* on page A8-32.
<const>	The immediate value to be added to the value obtained from <Rn>. See *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

The pre-UAL syntax ADD<c>S is equivalent to ADDS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ADD{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ。<Rn> に SP を指定した場合については、*ADD（SP + イミディエート）* P. A8-28 を参照して下さい。<Rn> に PC を指定した場合については、*ADR* P. A8-32 を参照して下さい。
<const>	<Rn> から取得された値に加算されるイミディエート値。値の範囲については、*ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

UAL 以前の構文の ADD<c>S は、ADDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.6 ADD (register)

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ADDS <Rd>,<Rn>,<Rm>                    Outside IT block.
ADD<c> <Rd>,<Rn>,<Rm>                  Inside IT block.
```

15 14 13	12 11	10	9	8 7 6	5 4 3	2 1 0
0 0 0	1 1	0	0	Rm	Rn	Rd

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7 if <Rdn> and <Rm> are both from R0-R7
ARMv4T, ARMv5T*, ARMv6*, ARMv7 otherwise

```
ADD<c> <Rdn>,<Rm>                      If <Rdn> is the PC, must be outside or last in IT block.
```

15 14 13	12 11	10	9	8	7	6 5 4	3 2 1	0
0 1 0	0 0	1	0	0	DN	Rm	Rdn	

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn);  n = d;  m = UInt(Rm);  setflags = FALSE;  (shift_t, shift_n) = (SRType_LSL, 0);
if n == 15 && m == 15 then UNPREDICTABLE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

```
ADD{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15 14 13	12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12 11	10 9 8 7	6 5 4	3 2 1 0
1 1 1	0 1	0 1	1 0 0 0	S	Rn	(0)	imm3	Rd	imm2 type	Rm

```
if Rd == '1111' && S == '1' then SEE CMN (register);
if Rn == '1101' then SEE ADD (SP plus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || n == 15 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADD{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
cond	0 0	0	0 1 0 0	S	Rn	Rd	imm5	type	0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
if Rn == '1101' then SEE ADD (SP plus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.6 ADD（レジスタ）

ADD（加算（レジスタ））は、レジスタの値と、オプションでシフトしたレジスタの値を加算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ADDS <Rd>,<Rn>,<Rm>                    IT ブロック外部
ADD<c> <Rd>,<Rn>,<Rm>                  IT ブロック内部
```

15 14 13	12 11	10	9	8 7 6	5 4 3	2 1 0
0 0 0	1 1	0	0	Rm	Rn	Rd

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7（<Rdn> と <Rm> の両方が R0 ～ R7 の場合）
ARMv4T、ARMv5T*、ARMv6*、ARMv7（上記以外の場合）

```
ADD<c> <Rdn>,<Rm>                      <Rdn> が PC の場合は、IT ブロックの外部に配置するか、
                                       IT ブロック内部の最後の命令にする必要があります。
```

15 14 13	12 11	10	9	8	7	6 5 4	3 2 1	0
0 1 0	0 0	1	0	0	DN	Rm	Rdn	

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn);  n = d;  m = UInt(Rm);  setflags = FALSE;  (shift_t, shift_n) = (SRType_LSL, 0);
if n == 15 && m == 15 then UNPREDICTABLE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

```
ADD{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15 14 13	12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12 11	10 9 8 7	6 5 4	3 2 1 0
1 1 1	0 1	0 1	1 0 0 0	S	Rn	(0)	imm3	Rd	imm2 type	Rm

```
if Rd == '1111' && S == '1' then SEE CMN (register);
if Rn == '1101' then SEE ADD (SP plus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || n == 15 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ADD{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
cond	0 0	0	0 1 0 0	S	Rn	Rd	imm5	type	0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
if Rn == '1101' then SEE ADD (SP plus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ADD{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register. If omitted, <Rd> is the same as <Rn> and encoding T2 is preferred to encoding T1 inside an IT block. If <Rd> is present, encoding T1 is preferred to encoding T2.

<Rn> — The first operand register. If <Rn> is SP, see *ADD (SP plus register)* on page A8-30.

<Rm> — The register that is optionally shifted and used as the second operand.

<shift> — The shift to apply to the value read from <Rm>. If present, only encoding T3 or A1 is permitted. If omitted, no shift is applied and any encoding is permitted. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

In Thumb assembly, inside an IT block, if ADD<c> <Rd>,<Rn>,<Rd> cannot be assembled using encoding T1, it is assembled using encoding T2 as though ADD<c> <Rd>,<Rn> had been written.

To prevent this happening, use the .W qualifier.

The pre-UAL syntax ADD<c>S is equivalent to ADDS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ADD{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ。省略した場合、<Rd> として <Rn> が使用され、IT ブロック内部でエンコード T1 よりもエンコード T2 の方が適しています。<Rd> を指定した場合は、エンコード T2 よりもエンコード T1 の方が適しています。

<Rn> — 最初のオペランドレジスタ。<Rn> が SP の場合は、*ADD (SP + レジスタ)* P. A8-30 を参照して下さい。

<Rm> — オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ

<shift> — <Rm> から読み出された値に適用されるシフト。指定した場合、エンコード T3 または A1 のみ許可されます。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

Thumb アセンブリでは、IT ブロック内部で、エンコード T1 を使用して ADD<c> <Rd>,<Rn>,<Rd> をアセンブルできない場合、ADD<c> <Rd>,<Rn> と記述されたものとして、エンコード T2 を使用してアセンブルされます。

これを避けるには、.W 修飾子を使用します。

UAL 以前の構文の ADD<c>S は、ADDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.7 ADD (register-shifted register)

Add (register-shifted register) adds a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADD{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 1 0 0	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.7 ADD（レジスタによりシフトしたレジスタ）

ADD（加算（レジスタによりシフトしたレジスタ））は、レジスタの値とレジスタによりシフトしたレジスタの値とを加算します。結果をデスティネーションレジスタに書き込み、オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ADD{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 1 0 0	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ADD{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The register that is shifted and used as the second operand.

<type> — The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR — Arithmetic shift right, encoded as type = 0b10
- LSL — Logical shift left, encoded as type = 0b00
- LSR — Logical shift right, encoded as type = 0b01
- ROR — Rotate right, encoded as type = 0b11.

<Rs> — The register whose bottom byte contains the amount to shift by.

The pre-UAL syntax ADD<c>S is equivalent to ADDS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ADD{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> — <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR — 算術右シフト。type = 0b10 としてエンコードされます。
- LSL — 論理左シフト。type = 0b00 としてエンコードされます。
- LSR — 論理右シフト。type = 0b01 としてエンコードされます。
- ROR — 右ローテート。type = 0b11 としてエンコードされます。

<Rs> — 最下位バイトにシフトのビット数を含むレジスタ

UAL 以前の構文の ADD<c>S は、ADDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.8 ADD (SP plus immediate)

This instruction adds an immediate value to the SP value, and writes the result to the destination register.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ADD<c> <Rd>,SP,#<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	Rd			imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm8:'00', 32);
```

Encoding T2 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ADD<c> SP,SP,#<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	imm7						

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

Encoding T3 ARMv6T2, ARMv7

```
ADD{S}<c>.W <Rd>,SP,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	1	1	0	1	0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
d = UInt(Rd);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv6T2, ARMv7

```
ADDW<c> <Rd>,SP,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	1	1	0	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADD{S}<c> <Rd>,SP,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	1	0	0	S	1	1	0	1	Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.8 ADD（SP + イミディエート）

ADD（加算（SP + イミディエート））は、イミディエート値を SP の値に加算して、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ADD<c> <Rd>,SP,#<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	Rd			imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm8:'00', 32);
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ADD<c> SP,SP,#<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	imm7						

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

エンコード T3 ARMv6T2、ARMv7

```
ADD{S}<c>.W <Rd>,SP,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	1	1	0	1	0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
d = UInt(Rd);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 15 then UNPREDICTABLE;
```

エンコード T4 ARMv6T2、ARMv7

```
ADDW<c> <Rd>,SP,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	1	1	0	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ADD{S}<c> <Rd>,SP,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	1	0	0	S	1	1	0	1	Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`ADD{S}<c><q> {<Rd>,} SP, #<const>`	All encodings permitted
`ADDW<c><q> {<Rd>,} SP, #<const>`	Only encoding T4 is permitted

where:

`S` If `S` is present, the instruction updates the flags. Otherwise, the flags are not updated.

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register. If omitted, `<Rd>` is SP.

`<const>` The immediate value to be added to the value obtained from SP. Values are multiples of 4 in the range 0-1020 for encoding T1, multiples of 4 in the range 0-508 for encoding T2 and any value in the range 0-4095 for encoding T4. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values for encodings T3 and A1.

When both 32-bit encodings are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the `ADDW` syntax).

The pre-UAL syntax `ADD<c>S` is equivalent to `ADDS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, imm32, '0');
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`ADD{S}<c><q> {<Rd>,} SP, #<const>`	すべてのエンコードが許可される
`ADDW<c><q> {<Rd>,} SP, #<const>`	エンコード T4 のみが許可される

各項目の説明については以下を参照して下さい。

`S` `S` を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ。省略した場合、`<Rd>` として SP が使用されます。

`<const>` SP から取得された値に加算されるイミディエート値。値の範囲は、エンコード T1 では 4 の倍数で 0 ～ 1020、エンコード T2 では 4 の倍数で 0 ～ 508、エンコード T4 では 0 ～ 4095 の任意の値です。エンコード T3 および A1 での値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

両方の 32 ビットエンコードが使用可能な場合、エンコード T4 よりもエンコード T3 の方が適しています（T4 エンコードが必要な場合は、`ADDW` 構文を使用して下さい）。

UAL 以前の構文の `ADD<c>S` は、`ADDS<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, imm32, '0');
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.9 ADD (SP plus register)

This instruction adds an optionally-shifted register value to the SP value, and writes the result to the destination register.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ADD<c> <Rdm>, SP, <Rdm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	DM	1	1	0	1	Rdm		

```
d = UInt(DM:Rdm);  m = UInt(DM:Rdm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ADD<c> SP,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	Rm				1	0	1

```
if Rm == '1101' then SEE encoding T1;
d = 13;  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T3 ARMv6T2, ARMv7

```
ADD{S}<c>.W <Rd>,SP,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	0	0	S	1	1	0	1	0	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 && (shift_t != SRType_LSL || shift_n > 3) then UNPREDICTABLE;
if d == 15 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADD{S}<c> <Rd>,SP,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	1	0	0	S	1	1	0	1	Rd				imm5				type			0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.9 ADD（SP + レジスタ）

ADD（加算（SP + レジスタ））は、オプションでシフトされるレジスタの値を SP の値に加算して、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ADD<c> <Rdm>, SP, <Rdm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	DM	1	1	0	1	Rdm		

```
d = UInt(DM:Rdm);  m = UInt(DM:Rdm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ADD<c> SP,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	Rm				1	0	1

```
if Rm == '1101' then SEE encoding T1;
d = 13;  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T3 ARMv6T2、ARMv7

```
ADD{S}<c>.W <Rd>,SP,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	0	0	S	1	1	0	1	0	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 && (shift_t != SRType_LSL || shift_n > 3) then UNPREDICTABLE;
if d == 15 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ADD{S}<c> <Rd>,SP,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	1	0	0	S	1	1	0	1	Rd				imm5				type			0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ADD{S}<c><q>  {<Rd>,} SP, <Rm>{, <shift>}
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register. This register can be SP. If omitted, <Rd> is SP. This register can be the PC, but if it is, encoding T3 is not permitted. Using the PC is deprecated.

<Rm> — The register that is optionally shifted and used as the second operand. This register can be the PC, but if it is, encoding T3 is not permitted. Using the PC is deprecated. This register can be SP in both ARM and Thumb instructions, but:

- the use of SP is deprecated
- when assembling for the Thumb instruction set, only encoding T1 is available and so the instruction can only be ADD SP,SP,SP.

<shift> — The shift to apply to the value read from <Rm>. If omitted, no shift is applied and any encoding is permitted. If present, only encoding T3 or A1 is permitted. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

In the Thumb instruction set, if <Rd> is SP or omitted, <shift> is only permitted to be omitted, LSL #1, LSL #2, or LSL #3.

The pre-UAL syntax ADD<c>S is equivalent to ADDS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(SP, shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ADD{S}<c><q>  {<Rd>,} SP, <Rm>{, <shift>}
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ。このレジスタとして SP を使用できます。省略した場合、<Rd> として SP が使用されます。このレジスタとして PC を使用できますが、使用した場合、エンコード T3 は許可されません。PC の使用は推奨されません。

<Rm> — オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ。このレジスタとして PC を使用できますが、使用した場合、エンコード T3 は許可されません。PC の使用は推奨されません。ARM 命令と Thumb 命令の両方で、このレジスタに SP を指定できますが、次の条件が適用されます。

- SP の使用は推奨されません。
- Thumb 命令セットのアセンブル時は、使用できるエンコードは T1 のみで、命令として ADD SP,SP,SP だけが許可されます。

<shift> — <Rm> から読み出された値に適用されるシフト。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。指定した場合、エンコード T3 または A1 のみ許可されます。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

Thumb 命令セットでは、<Rd> として SP を使用した場合、または Rd を省略した場合、<shift> のみ省略することが許可されます（LSL #1、LSL #2、または LSL #3）。

UAL 以前の構文の ADD<c>S は、ADDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(SP, shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.10 ADR

This instruction adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ADR<c> <Rd>,<label>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

Encoding T2 ARMv6T2, ARMv7

```
ADR<c>.W <Rd>,<label>
SUB <Rd>,PC,#0
```

<label> before current instruction
Special case for subtraction of zero

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	0	1	0	1	1	1	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = FALSE;
if BadReg(d) then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

```
ADR<c>.W <Rd>,<label>
```

<label> after current instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	1	1	1	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = TRUE;
if BadReg(d) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADR<c> <Rd>,<label>
```

<label> after current instruction

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	1	0	0	0	1	1	1	1	Rd				imm12											

```
d = UInt(Rd);  imm32 = ARMExpandImm(imm12);  add = TRUE;
```

Encoding A2 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADR<c> <Rd>,<label>
SUB <Rd>,PC,#0
```

<label> before current instruction
Special case for subtraction of zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	0	1	0	0	1	1	1	1	Rd				imm12											

```
d = UInt(Rd);  imm32 = ARMExpandImm(imm12);  add = FALSE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.10 ADR

ADR（PC 相対アドレスの生成）は、イミディエート値を PC の値に加算して、PC 相対アドレスを生成し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ADR<c> <Rd>,<label>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

エンコード T2 ARMv6T2、ARMv7

```
ADR<c>.W <Rd>,<label>
SUB <Rd>,PC,#0
```

<label> は現在の命令の前
0 の減算の特殊なケース

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	0	1	0	1	1	1	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = FALSE;
if BadReg(d) then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

```
ADR<c>.W <Rd>,<label>
```

<label> は現在の命令の後

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	1	1	1	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = TRUE;
if BadReg(d) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ADR<c> <Rd>,<label>
```

<label> は現在の命令の後

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	1	0	0	0	1	1	1	1	Rd				imm12											

```
d = UInt(Rd);  imm32 = ARMExpandImm(imm12);  add = TRUE;
```

エンコード A2 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ADR<c> <Rd>,<label>
SUB <Rd>,PC,#0
```

<label> は現在の命令の前
0 の減算の特殊なケース

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	0	1	0	0	1	1	1	1	Rd				imm12											

```
d = UInt(Rd);  imm32 = ARMExpandImm(imm12);  add = FALSE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`ADR<c><q> <Rd>, <label>`	Normal syntax
`ADD<c><q> <Rd>, PC, #<const>`	Alternative for encodings T1, T3, A1
`SUB<c><q> <Rd>, PC, #<const>`	Alternative for encoding T2, A2

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<label>` The label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC,4)` value of the `ADR` instruction to this label. Permitted values of the offset are:

Encoding T1

multiples of 4 in the range -1020 to 1020

Encodings T2 and T3

any value in the range -4095 to 4095

Encodings A1 and A2

plus or minus any of the constants described in *Modified immediate constants in ARM instructions* on page A5-9.

If the offset is zero or positive, encodings T1, T3, and A1 are permitted with `imm32` equal to the offset.

If the offset is negative, encodings T2 and A2 are permitted with `imm32` equal to minus the offset.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    if d == 15 then          // Can only occur for ARM encodings
        ALUWritePC(result);
    else
        R[d] = result;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`ADR<c><q> <Rd>, <label>`	通常の構文
`ADD<c><q> <Rd>, PC, #<const>`	エンコード T1、T3、A1 用の代替構文
`SUB<c><q> <Rd>, PC, #<const>`	エンコード T2 および A2 用の代替構文

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<label>` アドレスが `<Rd>` にロードされる命令またはリテラルデータ項目のラベル。アセンブラは、`ADR` 命令の `Align(PC,4)` の値からこのラベルまでのオフセットに必要な値を計算します。オフセットに許容される値の範囲は次のとおりです。

エンコード T1

4 の倍数で − 1020 〜 1020

エンコード T2 および T3

− 4095 〜 4095 の任意の値

エンコード A1 および A2

ARM 命令での修飾イミディエート定数 P. A5-9 で説明されている定数の正または負の値

オフセットが 0 または正の場合、エンコード T1、T3、A1 が許可され、`imm32` がオフセットに等しくなります。

オフセットが負の場合、エンコード T2 および A2 が許可され、`imm32` がオフセットの負の値に等しくなります。

代替構文では、通常の構文では指定できない 0 の減算など、オフセットおよびイミディエートオフセットの加算または減算を個別に指定することが許可されます。詳細については、*UAL 命令構文でのラベルの使用* P. A4-5 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    if d == 15 then          // Can only occur for ARM encodings
        ALUWritePC(result);
    else
        R[d] = result;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.11 AND (immediate)

This instruction performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
AND{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	0	0	S	Rn				0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE TST (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
AND{S}<c> <Rd>,<Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	0	0	0	S	Rn				Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.11 AND（イミディエート）

AND（論理積（イミディエート））は、レジスタの値とイミディエート値との間でビット単位の論理積を実行し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
AND{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	0	0	S	Rn				0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE TST (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
AND{S}<c> <Rd>,<Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	0	0	0	S	Rn				Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
AND{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<const>	The immediate value to be ANDed with the value obtained from <Rn>. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

The pre-UAL syntax AND<c>S is equivalent to ANDS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
AND{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<const>	<Rn> から取得された値との論理積が実行されるイミディエート値。値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

UAL 以前の構文の AND<c>S は、ANDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.12 AND (register)

This instruction performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ANDS    <Rdn>,<Rm>                    Outside IT block.
AND<c> <Rdn>,<Rm>                     Inside IT block.
```

15 14 13 12 11 10	9 8 7 6	5 4 3	2 1 0
0 1 0 0 0 0	0 0 0 0	Rm	Rdn

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
AND{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 0 0 0	S	Rn	(0)	imm3	Rd	imm2	type	Rm

```
if Rd == '1111' && S == '1' then SEE TST (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
AND{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 0	0	0 0 0 0	S	Rn	Rd	imm5	type		0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.12 AND（レジスタ）

AND（論理積（レジスタ））は、レジスタの値と、オプションとしてシフトしたレジスタの値との間でビット単位の論理積を実行し、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ANDS    <Rdn>,<Rm>                    IT ブロック外部
AND<c> <Rdn>,<Rm>                     IT ブロック内部
```

15 14 13 12 11 10	9 8 7 6	5 4 3	2 1 0
0 1 0 0 0 0	0 0 0 0	Rm	Rdn

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
AND{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 0 0 0	S	Rn	(0)	imm3	Rd	imm2	type	Rm

```
if Rd == '1111' && S == '1' then SEE TST (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
AND{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 0	0	0 0 0 0	S	Rn	Rd	imm5	type		0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
AND{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The register that is optionally shifted and used as the second operand.
<shift>	The shift to apply to the value read from <Rm>. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

In Thumb assembly:

- outside an IT block, if ANDS <Rd>,<Rn>,<Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ANDS <Rd>,<Rn> had been written
- inside an IT block, if AND<c> <Rd>,<Rn>,<Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though AND<c> <Rd>,<Rn> had been written.

To prevent either of these happening, use the .W qualifier.

The pre-UAL syntax AND<c>S is equivalent to ANDS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
AND{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ
<shift>	<Rm> から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。シフトとシフトのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

Thumb アセンブラでは、次の規則が適用されます。

- IT ブロック外部で、ANDS <Rd>,<Rn>,<Rd> の <Rd> と <Rn> の両方が R0 ～ R7 の範囲に含まれる場合、ANDS <Rd>,<Rn> と記述されたものとして、エンコード T1 を使用してアセンブルされます。
- IT ブロック内部で、AND<c> <Rd>,<Rn>,<Rd> の <Rd> と <Rn> の両方が R0 ～ R7 の範囲に含まれる場合、AND<c> <Rd>,<Rn> と記述されたものとして、エンコード T1 を使用してアセンブルされます。

これらの事態を避けるには、.W 修飾子を使用します。

UAL 以前の構文の AND<c>S は、ANDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.13 AND (register-shifted register)

This instruction performs a bitwise AND of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
AND{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 0 0 0	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.13 AND（レジスタによりシフトしたレジスタ）

AND（論理積（レジスタによりシフトしたレジスタ））は、レジスタの値と、レジスタによりシフトしたレジスタの値との間でビット単位の論理積を実行します。結果をデスティネーションレジスタに書き込み、オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
AND{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 0 0 0	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
AND{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The register that is shifted and used as the second operand.

<type> — The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR — Arithmetic shift right, encoded as type = 0b10
- LSL — Logical shift left, encoded as type = 0b00
- LSR — Logical shift right, encoded as type = 0b01
- ROR — Rotate right, encoded as type = 0b11.

<Rs> — The register whose bottom byte contains the amount to shift by.

The pre-UAL syntax AND<c>S is equivalent to ANDS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
AND{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> — <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR — 算術右シフト。type = 0b10 としてエンコードされます。
- LSL — 論理左シフト。type = 0b00 としてエンコードされます。
- LSR — 論理右シフト。type = 0b01 としてエンコードされます。
- ROR — 右ローテート。type = 0b11 としてエンコードされます。

<Rs> — 最下位バイトにシフトのビット数を含むレジスタ

UAL 以前の構文の AND<c>S は、ANDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.14 ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ASRS <Rd>,<Rm>,#<imm>                    Outside IT block.
ASR<c> <Rd>,<Rm>,#<imm>                  Inside IT block.
```

15 14 13	12 11	10 9 8 7 6	5 4 3	2 1 0
0 0 0	1 0	imm5	Rm	Rd

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('10', imm5);
```

Encoding T2 ARMv6T2, ARMv7

```
ASR{S}<c>.W <Rd>,<Rm>,#<imm>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0	1 0 1	0 0 1 0	S 1 1 1 1	(0)	imm3	Rd	imm2	1 0 Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('10', imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ASR{S}<c> <Rd>,<Rm>,#<imm>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5 4	3 2 1 0
cond	0 0	0	1 1 0 1	S	(0) (0) (0) (0)	Rd	imm5	1 0 0	Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('10', imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.14 ASR（イミディエート）

ASR（算術右シフト（イミディエート））は、レジスタの値のビットを、イミディエート値で指定されたビット数だけ右にシフト（空いたビットには、符号ビットがコピーされます）して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ASRS <Rd>,<Rm>,#<imm>                    IT ブロック外部
ASR<c> <Rd>,<Rm>,#<imm>                  IT ブロック内部
```

15 14 13	12 11	10 9 8 7 6	5 4 3	2 1 0
0 0 0	1 0	imm5	Rm	Rd

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('10', imm5);
```

エンコード T2 ARMv6T2、ARMv7

```
ASR{S}<c>.W <Rd>,<Rm>,#<imm>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0	1 0 1	0 0 1 0	S 1 1 1 1	(0)	imm3	Rd	imm2	1 0 Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('10', imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ASR{S}<c> <Rd>,<Rm>,#<imm>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5 4	3 2 1 0
cond	0 0	0	1 1 0 1	S	(0) (0) (0) (0)	Rd	imm5	1 0 0	Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('10', imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ASR{S}<c><q>  {<Rd>,} <Rm>, #<imm>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rm>	The first operand register.
<imm>	The shift amount, in the range 1 to 32. See *Shifts applied to a register* on page A8-10.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_ASR, shift_n, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ASR{S}<c><q>  {<Rd>,} <Rm>, #<imm>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rm>	最初のオペランドレジスタ
<imm>	1 ～ 32 の範囲のシフト量。詳細については、*レジスタに適用されるシフト* P. A8- 10 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_ASR, shift_n, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.15 ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ASRS   <Rdn>,<Rm>
ASR<c> <Rdn>,<Rm>
```

Outside IT block.
Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

Encoding T2 ARMv6T2, ARMv7

```
ASR{S}<c>.W <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	S	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ASR{S}<c> <Rd>,<Rn>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				Rm				0	1	0	1	Rn			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.15 ASR（レジスタ）

ASR（算術右シフト（レジスタ））は、レジスタの値を可変のビット数だけ右にシフト（空いたビットには、符号ビットがコピーされます）し、結果をデスティネーションレジスタに書き込みます。ローテートするビット数は、レジスタの最下位バイトから読み出されます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ASRS   <Rdn>,<Rm>
ASR<c> <Rdn>,<Rm>
```

IT ブロック外部
IT ブロック内部

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

エンコード T2 ARMv6T2、ARMv7

```
ASR{S}<c>.W <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	S	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ASR{S}<c> <Rd>,<Rn>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				Rm				0	1	0	1	Rn			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ASR{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_ASR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ASR{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	最下位バイトにシフトのビット数を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_ASR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.16 B

Branch causes a branch to a target address.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

B<c> <label> — Not permitted in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	cond				imm8							

```
if cond == '1110' then UNDEFINED;
if cond == '1111' then SEE SVC;
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

Encoding T2 ARMv4T, ARMv5T*, ARMv6*, ARMv7

B<c> <label> — Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	imm11										

```
imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

B<c>.W <label> — Not permitted in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	cond				imm6						1	0	J1	0	J2	imm11										

```
if cond<3:1> == '111' then SEE "Related encodings";
imm32 = SignExtend(S:J2:J1:imm6:imm11:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

Encoding T4 ARMv6T2, ARMv7

B<c>.W <label> — Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	0	J1	1	J2	imm11										

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

B<c> <label>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	1	0	imm24																							

```
imm32 = SignExtend(imm24:'00', 32);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.16 B

B（分岐）は、ターゲットアドレスへの分岐を発生させます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

B<c> <label> — IT ブロック内部では許可されない

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	cond				imm8							

```
if cond == '1110' then UNDEFINED;
if cond == '1111' then SEE SVC;
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

B<c> <label> — IT ブロック外部または IT ブロック内部の最後

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	imm11										

```
imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

B<c>.W <label> — IT ブロック内部では許可されない

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	cond				imm6						1	0	J1	0	J2	imm11										

```
if cond<3:1> == '111' then SEE "Related encodings";
imm32 = SignExtend(S:J2:J1:imm6:imm11:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

エンコード T4 ARMv6T2、ARMv7

B<c>.W <label> — IT ブロック外部または IT ブロック内部の最後

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	0	J1	1	J2	imm11										

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

B<c> <label>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	1	0	imm24																							

```
imm32 = SignExtend(imm24:'00', 32);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Related encodings See *Branches and miscellaneous control* on page A6-20

Assembler syntax

```
B<c><q>  <label>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

Note

Encodings T1 and T3 are conditional in their own right, and do not require an IT instruction to make them conditional.

For encodings T1 and T3, <c> must not be AL or omitted. The 4-bit encoding of the condition is placed in the instruction and not in a preceding IT instruction, and the instruction must not be in an IT block. As a result, encodings T1 and T2 are never both available to the assembler, nor are encodings T3 and T4.

<label> The label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset.

Permitted offsets are:

Encoding T1	Even numbers in the range –256 to 254
Encoding T2	Even numbers in the range –2048 to 2046
Encoding T3	Even numbers in the range –1048576 to 1048574
Encoding T4	Even numbers in the range –16777216 to 16777214
Encoding A1	Multiples of 4 in the range –33554432 to 33554428.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

関連エンコード *分岐およびその他の制御* P. A6- 20 を参照して下さい。

アセンブラ構文

```
B<c><q>  <label>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

注

エンコード T1 および T3 はそれ自体が条件付きで、IT 命令を使用して条件付きにする必要はありません。

エンコード T1 および T3 では、<c> に AL を指定できず、省略もできません。条件の 4 ビットエンコードは、先行する IT 命令内ではなく、この命令に含まれ、この命令は IT ブロック内で使用できません。結果として、エンコード T1 および T2 の両方、またはエンコード T3 および T4 の両方をアセンブラで同時に使用することはできません。

<label> 分岐先の命令のラベル。アセンブラは、B 命令の PC の値からこのラベルまでのオフセットに必要な値を計算し、imm32 をそのオフセットに設定するエンコードを選択します。

許可されるオフセットは次のとおりです。

エンコード T1	− 256 ～ 254 の範囲の偶数
エンコード T2	− 2048 ～ 2046 の範囲の偶数
エンコード T3	− 1048576 ～ 1048574 の範囲の偶数
エンコード T4	− 16777216 ～ 16777214 の範囲の偶数
エンコード A1	4 の倍数で − 33554432 ～ 33554428

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.17 BFC

Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

Encoding T1 ARMv6T2, ARMv7

```
BFC<c> <Rd>,#<lsb>,#<width>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	1	0	1	1	1	1	0	imm3			Rd				imm2		(0)	msb				

```
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if BadReg(d) then UNPREDICTABLE;
```

Encoding A1 ARMv6T2, ARMv7

```
BFC<c> <Rd>,#<lsb>,#<width>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	1	1	1	0	msb					Rd				lsb					0	0	1	1	1	1	1

```
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.17 BFC

BFC（ビットフィールド クリア）は、レジスタ内の任意の位置にある、隣接する任意の数のビットをクリアします。レジスタ内の他のビットには影響を与えません。

エンコード T1 ARMv6T2、ARMv7

```
BFC<c> <Rd>,#<lsb>,#<width>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	1	0	1	1	1	1	0	imm3			Rd				imm2		(0)	msb				

```
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if BadReg(d) then UNPREDICTABLE;
```

エンコード A1 ARMv6T2、ARMv7

```
BFC<c> <Rd>,#<lsb>,#<width>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	1	1	1	0	msb					Rd				lsb					0	0	1	1	1	1	1

```
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
BFC<c><q>  <Rd>, #<lsb>, #<width>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<lsb> The least significant bit that is to be cleared, in the range 0 to 31. This determines the required value of lsbit.

<width> The number of bits to be cleared, in the range 1 to 32-<lsb>. The required value of msbit is <lsb>+<width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = Replicate('0', msbit-lsbit+1);
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
BFC<c><q>  <Rd>, #<lsb>, #<width>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<lsb> クリアする最下位ビットで、範囲は 0 ～ 31 です。これにより、lsbit に必要な値が決定されます。

<width> クリアするビット数で、範囲は 1 ～ 32 − <lsb> です。msbit に必要な値は <lsb> + <width> - 1 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = Replicate('0', msbit-lsbit+1);
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.18 BFI

Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

Encoding T1 ARMv6T2, ARMv7

```
BFI<c> <Rd>,<Rn>,#<lsb>,#<width>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	1	0	Rn				0	imm3			Rd				imm2		(0)		msb			

```
if Rn == '1111' then SEE BFC;
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if BadReg(d) || n == 13 then UNPREDICTABLE;
```

Encoding A1 ARMv6T2, ARMv7

```
BFI<c> <Rd>,<Rn>,#<lsb>,#<width>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	1	1	1	0	msb					Rd				lsb					0	0	1	Rn			

```
if Rn == '1111' then SEE BFC;
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.18 BFI

BFI（ビットフィールド挿入）は、ソースレジスタから任意の数の低位ビットを、デスティネーションレジスタの任意の位置にある同じ数の隣接するビットにコピーします。

エンコード T1 ARMv6T2、ARMv7

```
BFI<c> <Rd>,<Rn>,#<lsb>,#<width>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	1	0	Rn				0	imm3			Rd				imm2		(0)		msb			

```
if Rn == '1111' then SEE BFC;
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if BadReg(d) || n == 13 then UNPREDICTABLE;
```

エンコード A1 ARMv6T2、ARMv7

```
BFI<c> <Rd>,<Rn>,#<lsb>,#<width>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	1	1	1	0	msb					Rd				lsb					0	0	1	Rn			

```
if Rn == '1111' then SEE BFC;
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
BFI<c><q>  <Rd>, <Rn>, #<lsb>, #<width>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The source register.

<lsb> The least significant destination bit, in the range 0 to 31. This determines the required value of lsbit.

<width> The number of bits to be copied, in the range 1 to 32-<lsb>. The required value of msbit is <lsb>+<width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = R[n]<(msbit-lsbit):0>;
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
BFI<c><q>  <Rd>, <Rn>, #<lsb>, #<width>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> ソースレジスタ

<lsb> 0 ～ 31 の範囲の最下位デスティネーションビット。これにより、lsbit に必要な値が決定されます。

<width> コピーするビット数で、範囲は 1 ～ 32 – <lsb> です。msbit に必要な値は <lsb> + <width> - 1 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = R[n]<(msbit-lsbit):0>;
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.19 BIC (immediate)

Bitwise Bit Clear (immediate) performs a bitwise AND of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv6T2, ARMv7

```
BIC{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	0	1	S	Rn				0	imm3			Rd				imm8							

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
BIC{S}<c> <Rd>,<Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	1	1	0	S	Rn				Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.19 BIC（イミディエート）

BIC（ビット単位のビットクリア（イミディエート））は、レジスタの値と、イミディエート値の補数とのビット単位の論理積を実行し、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

```
BIC{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	0	1	S	Rn				0	imm3			Rd				imm8							

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
BIC{S}<c> <Rd>,<Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	1	1	0	S	Rn				Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
BIC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The register that contains the operand.
<const>	The immediate value to be bitwise inverted and ANDed with the value obtained from <Rn>. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

The pre-UAL syntax BIC<c>S is equivalent to BICS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND NOT(imm32);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
BIC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	オペランドを含むレジスタ
<const>	ビット単位で反転され、<Rn> から取得された値との論理積が実行されるイミディエート値。値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

UAL 以前の構文の BIC<c>S は、BICS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND NOT(imm32);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.20 BIC (register)

Bitwise Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
BICS    <Rdn>,<Rm>                    Outside IT block.
BIC<c> <Rdn>,<Rm>                     Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
BIC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	0	1	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
BIC{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	1	1	0	S	Rn			Rd				imm5					type		0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.20 BIC（レジスタ）

BIC（ビット単位のビットクリア（レジスタ））は、レジスタの値と、オプションでシフトされるレジスタの値の補数とでビット単位の論理積を実行し、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
BICS    <Rdn>,<Rm>                    ITブロック外部
BIC<c> <Rdn>,<Rm>                     ITブロック内部
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
BIC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	0	1	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
BIC{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	1	1	0	S	Rn			Rd				imm5					type		0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
BIC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The register that is optionally shifted and used as the second operand.
<shift>	The shift to apply to the value read from <Rm>. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

The pre-UAL syntax BIC<c>S is equivalent to BICS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND NOT(shifted);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
BIC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ
<shift>	<Rm> から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

UAL 以前の構文の BIC<c>S は、BICS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND NOT(shifted);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.21 BIC (register-shifted register)

Bitwise Bit Clear (register-shifted register) performs a bitwise AND of a register value and the complement of a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
BIC{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 1 1 0	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.21 BIC（レジスタによりシフトしたレジスタ）

BIC（ビット単位のビットクリア（レジスタによりシフトしたレジスタ））は、レジスタの値と、レジスタによりシフトしたレジスタの値の補数とのビット単位の論理積を実行します。結果をデスティネーションレジスタに書き込み、オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
BIC{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 1 1 0	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
BIC{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The register that is shifted and used as the second operand.

<type> — The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR — Arithmetic shift right, encoded as type = 0b10
- LSL — Logical shift left, encoded as type = 0b00
- LSR — Logical shift right, encoded as type = 0b01
- ROR — Rotate right, encoded as type = 0b11.

<Rs> — The register whose bottom byte contains the amount to shift by.

The pre-UAL syntax BIC<c>S is equivalent to BICS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
BIC{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> — <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR — 算術右シフト。type = 0b10 としてエンコードされます。
- LSL — 論理左シフト。type = 0b00 としてエンコードされます。
- LSR — 論理右シフト。type = 0b01 としてエンコードされます。
- ROR — 右ローテート。type = 0b11 としてエンコードされます。

<Rs> — 最下位バイトにシフトのビット数を含むレジスタ

UAL 以前の構文の BIC<c>S は、BICS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.22 BKPT

Breakpoint causes a software breakpoint to occur.

Breakpoint is always unconditional, even when inside an IT block.

Encoding T1 ARMv5T*, ARMv6*, ARMv7

```
BKPT #<imm8>
```

15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 0 1 1	1 1 1 0	imm8

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly only and is ignored by hardware.
```

Encoding A1 ARMv5T*, ARMv6*, ARMv7

```
BKPT #<imm16>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1 0 0 1 0	imm12	0 1 1 1	imm4

```
imm32 = ZeroExtend(imm12:imm4, 32);
// imm32 is for assembly/disassembly only and is ignored by hardware.

if cond != '1110' then UNPREDICTABLE;  // BKPT must be encoded with AL condition
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.22 BKPT

BKPT（ブレークポイント）は、ソフトウェアブレークポイントを引き起こします。

ブレークポイントは常に無条件です（IT ブロック内であっても）。

エンコード T1 ARMv5T*、ARMv6*、ARMv7

```
BKPT #<imm8>
```

15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 0 1 1	1 1 1 0	imm8

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly only and is ignored by hardware.
```

エンコード A1 ARMv5T*、ARMv6*、ARMv7

```
BKPT #<imm16>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1 0 0 1 0	imm12	0 1 1 1	imm4

```
imm32 = ZeroExtend(imm12:imm4, 32);
// imm32 is for assembly/disassembly only and is ignored by hardware.

if cond != '1110' then UNPREDICTABLE;  // BKPT must be encoded with AL condition
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
BKPT<q>  #<imm>
```

where:

<q> See *Standard assembler syntax fields* on page A8-7. A BKPT instruction must be unconditional.

<imm> Specifies a value that is stored in the instruction, in the range 0-255 for a Thumb instruction or 0-65535 for an ARM instruction. This value is ignored by the processor, but can be used by a debugger to store more information about the breakpoint.

Operation

```
EncodingSpecificOperations();
BKPTInstrDebugEvent();
```

Exceptions

Prefetch Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
BKPT<q>  #<imm>
```

各項目の説明については以下を参照して下さい。

<q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。BKPT は常に無条件命令です。

<imm> 命令に格納される値を指定します。値の範囲は、Thumb 命令では 0 〜 255、ARM 命令では 0 〜 65535 です。この値は、プロセッサでは無視されますが、デバッガでブレークポイントに関する詳細情報の保持に使用できます。

動作

```
EncodingSpecificOperations();
BKPTInstrDebugEvent();
```

例外

プリフェッチアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.23 BL, BLX (immediate)

Branch with Link calls a subroutine at a PC-relative address.

Branch with Link and Exchange Instruction Sets (immediate) calls a subroutine at a PC-relative address, and changes instruction set from ARM to Thumb, or from Thumb to ARM.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7 if J1 == J2 == 1
ARMv6T2, ARMv7 otherwise

```
BL<c> <label>                                                    Outside or last in IT block
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	1	J1	1	J2	imm11										

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
toARM = FALSE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T2 ARMv5T*, ARMv6*, ARMv7 if J1 == J2 == 1
ARMv6T2, ARMv7 otherwise

```
BLX<c> <label>                                                   Outside or last in IT block
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10H										1	1	J1	0	J2	imm10L										0

```
if CurrentInstrSet() == InstrSet_ThumbEE then UNDEFINED;
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10H:imm10L:'00', 32);
toARM = TRUE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
BL<c> <label>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	1	1	imm24																							

```
imm32 = SignExtend(imm24:'00', 32);  toARM = TRUE;
```

Encoding A2 ARMv5T*, ARMv6*, ARMv7

```
BLX <label>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	H	imm24																							

```
imm32 = SignExtend(imm24:H:'0', 32);  toARM = FALSE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.23 BL、BLX（イミディエート）

BL（リンク付き分岐）は、PC 相対アドレスでサブルーチンを呼び出します。

BLX（リンク付き分岐と命令セット状態遷移（イミディエート））は、PC 相対アドレスでサブルーチンを呼び出し、命令セットを ARM から Thumb に、または Thumb から ARM に変更します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7（J1 == J2 == 1 の場合）
ARMv6T2、ARMv7（その他の場合）

```
BL<c> <label>                                                    IT ブロック外部または IT ブロック
                                                                 内部の最後
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	1	J1	1	J2	imm11										

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
toARM = FALSE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード T2 ARMv5T*、ARMv6*、ARMv7（J1 == J2 == 1 の場合）
ARMv6T2、ARMv7（その他の場合）

```
BLX<c> <label>                                                   IT ブロック外部または IT ブロック
                                                                 内部の最後
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10H										1	1	J1	0	J2	imm10L										0

```
if CurrentInstrSet() == InstrSet_ThumbEE then UNDEFINED;
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10H:imm10L:'00', 32);
toARM = TRUE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
BL<c> <label>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	1	1	imm24																							

```
imm32 = SignExtend(imm24:'00', 32);  toARM = TRUE;
```

エンコード A2 ARMv5T*、ARMv6*、ARMv7

```
BLX <label>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	H	imm24																							

```
imm32 = SignExtend(imm24:H:'0', 32);  toARM = FALSE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
BL{X}<c><q>  <label>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7. An ARM BLX (immediate) instruction must be unconditional.

X If present, specifies a change of instruction set (from ARM to Thumb or from Thumb to ARM). If X is omitted, the processor remains in the same state. For ThumbEE code, specifying X is not permitted.

<label> The label of the instruction that is to be branched to.

For BL (encodings T1, A1), the assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range –16777216 to 16777214 (Thumb) or multiples of 4 in the range –33554432 to 33554428 (ARM).

For BLX (encodings T2, A2), the assembler calculates the required value of the offset from the Align(PC,4) value of the BLX instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are multiples of 4 in the range –16777216 to 16777212 (Thumb) or even numbers in the range –33554432 to 33554430 (ARM).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet == InstrSet_ARM then
        next_instr_addr = PC - 4;
        LR = next_instr_addr;
    else
        next_instr_addr = PC;
        LR = next_instr_addr<31:1> : '1';
    if toARM then
        SelectInstrSet(InstrSet_ARM);
        BranchWritePC(Align(PC,4) + imm32);
    else
        SelectInstrSet(InstrSet_Thumb);
        BranchWritePC(PC + imm32);
```

Exceptions

None.

Branch range before ARMv6T2

Before ARMv6T2, J1 and J2 in encodings T1 and T2 were both 1, resulting in a smaller branch range. The instructions could be executed as two separate 16-bit instructions, as described in *BL and BLX (immediate) instructions, before ARMv6T2* on page AppxG-4.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
BL{X}<c><q>  <label>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。.ARM BLX（イミディエート）は常に無条件命令です。

X 指定した場合、命令セットの変更（ARM から Thumb、または Thumb から ARM への）を指定します。省略した場合、プロセッサは同じ状態にとどまります。ThumbEE コードの場合、X を指定することはできません。

<label> 分岐先の命令のラベル。

BL（エンコード T1、A1）の場合、アセンブラは、BL 命令の PC の値からこのラベルまでのオフセットに必要な値を計算して、imm32 をそのオフセットに設定するエンコードを選択します。許可されるオフセットは、- 16777216 ～ 16777214 の範囲の偶数（Thumb の場合）か、– 33554432 ～ 33554428 の範囲の 4 の倍数（ARM の場合）です。

BLX（エンコード T2、A2）の場合、アセンブラは、BLX 命令の値の Align(PC,4) からこのラベルまでのオフセットに必要な値を計算して、imm32 をそのオフセットに設定するエンコードを選択します。許可されるオフセットは、- 16777216 ～ 16777212 の範囲の 4 の倍数（Thumb の場合）か、– 33554432 ～ 33554430 の範囲の偶数（ARM の場合）です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet == InstrSet_ARM then
        next_instr_addr = PC - 4;
        LR = next_instr_addr;
    else
        next_instr_addr = PC;
        LR = next_instr_addr<31:1> : '1';
    if toARM then
        SelectInstrSet(InstrSet_ARM);
        BranchWritePC(Align(PC,4) + imm32);
    else
        SelectInstrSet(InstrSet_Thumb);
        BranchWritePC(PC + imm32);
```

例外

なし

ARMv6T2 以前の分岐範囲

ARMv6T2 以前は、エンコード T1 および T2 の J1 および J2 はいずれも 1 で、結果的に分岐範囲が制限されています。*ARM v 6T2 以前の BL および BLX（イミディエート）命令* P. 付録 G-4 で説明されているように、この命令を 2 つの異なる 16 ビット命令として実行することができます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.24 BLX (register)

Branch with Link and Exchange (register) calls a subroutine at an address and instruction set specified by a register.

Encoding T1 ARMv5T*, ARMv6*, ARMv7

```
BLX<c> <Rm>
```

Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	1	Rm				(0)	(0)	(0)

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding A1 ARMv5T*, ARMv6*, ARMv7

```
BLX<c> <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	1	0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	1	Rm			

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.24 BLX（レジスタ）

BLX（リンク付き分岐と状態遷移（レジスタ））は、レジスタで指定されたアドレスにある、指定された命令セットのサブルーチンを呼び出します。

エンコード T1 ARMv5T*、ARMv6*、ARMv7

```
BLX<c> <Rm>
```

IT ブロック外部または IT ブロック内部の最後

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	1	Rm				(0)	(0)	(0)

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード A1 ARMv5T*、ARMv6*、ARMv7

```
BLX<c> <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	1	0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	1	Rm			

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
BLX<c><q>  <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rm> The register that contains the branch target address and instruction set selection bit.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet() == InstrSet_ARM then
        next_instr_addr = PC - 4;
        LR = next_instr_addr;
    else
        next_instr_addr = PC - 2;
        LR = next_instr_addr<31:1> : '1';
    BXWritePC(R[m]);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
BLX<c><q>  <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rm> 分岐先アドレスと命令セット選択ビットを含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet() == InstrSet_ARM then
        next_instr_addr = PC - 4;
        LR = next_instr_addr;
    else
        next_instr_addr = PC - 2;
        LR = next_instr_addr<31:1> : '1';
    BXWritePC(R[m]);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.25 BX

Branch and Exchange causes a branch to an address and instruction set specified by a register.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
BX<c> <Rm>
```

Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0	Rm				(0)	(0)	(0)

```
m = UInt(Rm);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding A1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
BX<c> Rm
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	1	0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	0	1	Rm			

```
m = UInt(Rm);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.25 BX

BX（分岐と状態遷移）は、レジスタで指定されたアドレスにある、指定された命令セットへの分岐を発生させます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
BX<c> <Rm>
```

IT ブロック外部または IT ブロック内部の最後

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0	Rm				(0)	(0)	(0)

```
m = UInt(Rm);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード A1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
BX<c> Rm
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	1	0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	0	1	Rm			

```
m = UInt(Rm);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
BX<c><q>  <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rm> The register that contains the branch target address and instruction set selection bit. The PC can be used.

Note

If <Rm> is the PC in a Thumb instruction at a non word-aligned address, it results in UNPREDICTABLE behavior because the address passed to the BXWritePC() pseudocode function has bits<1:0> = '10'.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m]);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
BX<c><q>  <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rm> 分岐先アドレスと命令セット選択ビットを含むレジスタ。このレジスタには PC を指定できます。

注

<Rm> が、ワードアラインドでないアドレスの Thumb 命令の PC である場合、BXWritePC() 擬似関数に渡されるアドレスのビット 〈1:0〉 = '10' なため、どのような動作になるか予測不能です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m]);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.26 BXJ

Branch and Exchange Jazelle attempts to change to Jazelle state. If the attempt fails, it branches to an address and instruction set specified by a register as though it were a BX instruction.

Encoding T1 ARMv6T2, ARMv7

BXJ<c> <Rm> Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	0	0	Rm				1	0	(0)	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
m = UInt(Rm);
if BadReg(m) then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding A1 ARMv5TEJ, ARMv6*, ARMv7

BXJ<c> <Rm>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	1	0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	0	Rm			

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.26 BXJ

BXJ（分岐と Jazelle 状態遷移）は、Jazelle 状態への変更を試みます。この試みが失敗した場合、BX 命令の場合と同様に、レジスタで指定されたアドレスと命令セットに分岐します。

エンコード T1 ARMv6T2、ARMv7

BXJ<c> <Rm> IT ブロック外部または IT ブロック内部の最後

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	0	0	Rm				1	0	(0)	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
m = UInt(Rm);
if BadReg(m) then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード A1 ARMv5TEJ、ARMv6*、ARMv7

BXJ<c> <Rm>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	1	0	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	0	Rm			

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
BXJ<c><q>  <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rm> The register that specifies the branch target address and instruction set selection bit to be used if the attempt to switch to Jazelle state fails.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if JMCR.JE == '0' || CurrentInstrSet() == InstrSet_ThumbEE then
        BXWritePC(R[m]);
    else
        if JazelleAcceptsExecution() then
            SwitchToJazelleExecution();
        else
            SUBARCHITECTURE_DEFINED handler call;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
BXJ<c><q>  <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rm> Jazelle 状態への切り替えの試みが失敗した場合に使用される、分岐先アドレスと命令セット選択ビットを指定するレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if JMCR.JE == '0' || CurrentInstrSet() == InstrSet_ThumbEE then
        BXWritePC(R[m]);
    else
        if JazelleAcceptsExecution() then
            SwitchToJazelleExecution();
        else
            SUBARCHITECTURE_DEFINED handler call;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.27 CBNZ, CBZ

Compare and Branch on Nonzero and Compare and Branch on Zero compare the value in a register with zero, and conditionally branch forward a constant value. They do not affect the condition flags.

Encoding T1 ARMv6T2, ARMv7

```
CB{N}Z <Rn>,<label>
```

Not permitted in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	op	0	i	1	imm5					Rn		

```
n = UInt(Rn);  imm32 = ZeroExtend(i:imm5:'0', 32);  nonzero = (op == '1');
if InITBlock() then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.27 CBNZ、CBZ

CBNZ（比較して非 0 で分岐）および CBZ（比較して 0 で分岐）は、レジスタ内の値と 0 を比較して、条件付きで定数値分の前方分岐を実行します。条件フラグには影響を与えません。

エンコード T1 ARMv6T2、ARMv7

```
CB{N}Z <Rn>,<label>
```

IT ブロック内部では許可されない

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	op	0	i	1	imm5					Rn		

```
n = UInt(Rn);  imm32 = ZeroExtend(i:imm5:'0', 32);  nonzero = (op == '1');
if InITBlock() then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
CB{N}Z<q>  <Rn>, <label>
```

where:

N — If specified, causes the branch to occur when the contents of <Rn> are nonzero (encoded as op = 1). If omitted, causes the branch to occur when the contents of <Rn> are zero (encoded as op = 0).

<q> — See *Standard assembler syntax fields* on page A8-7. A CBZ or CBNZ instruction must be unconditional.

<Rn> — The operand register.

<label> — The label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the CB{N}Z instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range 0 to 126.

Operation

```
EncodingSpecificOperations();
if nonzero ^ IsZero(R[n]) then
    BranchWritePC(PC + imm32);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
CB{N}Z<q>  <Rn>, <label>
```

各項目の説明については以下を参照して下さい。

N — 指定した場合、<Rn> の内容が 0 以外である（op = 1 としてエンコードされる）ときに分岐が発生します。省略した場合、<Rn> が 0 である（op = 0 としてエンコードされる）ときに分岐が発生します。

<q> — *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。CBZ または CBNZ は常に無条件命令です。

<Rn> — オペランドレジスタ

<label> — 分岐先の命令のラベル。アセンブラは、CB{N}Z 命令の PC 値からこのラベルまでのオフセットに必要な値を計算して、imm32 をそのオフセットに設定するエンコードを選択します。オフセットに許容される値の範囲は、0 ～ 126 までの偶数です。

動作

```
EncodingSpecificOperations();
if nonzero ^ IsZero(R[n]) then
    BranchWritePC(PC + imm32);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.28 CDP, CDP2

Coprocessor Data Processing tells a coprocessor to perform an operation that is independent of ARM core registers and memory. If no coprocessor can execute the instruction, an Undefined Instruction exception is generated.

This is a generic coprocessor instruction. Some of the fields have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer. These fields are the opc1, opc2, CRd, CRn, and CRm fields.

For more information about the coprocessors see *Coprocessor support* on page A2-68.

Encoding T1 / A1 ARMv6T2, ARMv7 for encoding T1
ARMv4*, ARMv5T*, ARMv6*, ARMv7 for encoding A1

```
CDP<c> <coproc>,<opc1>,<CRd>,<CRn>,<CRm>,<opc2>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0	1 1 1 0	opc1	CRn	CRd	coproc	opc2	0 CRm

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	1 1 1 0	opc1	CRn	CRd	coproc	opc2	0 CRm

```
if coproc == '101x' then SEE "VFP instructions";
cp = UInt(coproc);
```

Encoding T2 / A2 ARMv6T2, ARMv7 for encoding T2
ARMv5T*, ARMv6*, ARMv7 for encodingA2

```
CDP2<c> <coproc>,<opc1>,<CRd>,<CRn>,<CRm>,<opc2>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	1 1 1 0	opc1	CRn	CRd	coproc	opc2	0 CRm

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	1 1 1 0	opc1	CRn	CRd	coproc	opc2	0 CRm

```
cp = UInt(coproc);
```

VFP instructions See *VFP data-processing instructions* on page A7-24

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.28 CDP、CDP2

CDP、CDP2（コプロセッサデータ処理）は、コプロセッサに対して、ARM コアレジスタやメモリと独立して動作を実行するように指示します。コプロセッサが命令を実行できない場合は、未定義命令例外が発生します。

これは、汎用のコプロセッサ命令です。一部のフィールドの機能はアーキテクチャで定義されていないため、コプロセッサ命令セットの設計者が自由に使用することができます。これらのフィールドは、opc1、opc2、CRd、CRn、CRm です。

コプロセッサの詳細については、*コプロセッサのサポート* P. A2-68 を参照して下さい。

エンコード T1/A1 ARMv6T2、ARMv7（エンコード T1 の場合）
ARMv4*、ARMv5T*、ARMv6*、ARMv7（エンコード A1 の場合）

```
CDP<c> <coproc>,<opc1>,<CRd>,<CRn>,<CRm>,<opc2>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0	1 1 1 0	opc1	CRn	CRd	coproc	opc2	0 CRm

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	1 1 1 0	opc1	CRn	CRd	coproc	opc2	0 CRm

```
if coproc == '101x' then SEE "VFP instructions";
cp = UInt(coproc);
```

エンコード T2/A2 ARMv6T2、ARMv7（エンコード T2 の場合）
ARMv5T*、ARMv6*、ARMv7（エンコード A2 の場合）

```
CDP2<c> <coproc>,<opc1>,<CRd>,<CRn>,<CRm>,<opc2>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	1 1 1 0	opc1	CRn	CRd	coproc	opc2	0 CRm

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	1 1 1 0	opc1	CRn	CRd	coproc	opc2	0 CRm

```
cp = UInt(coproc);
```

VFP 命令 *VFP のデータ処理命令* P. A7-24 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
CDP{2}<c><q>  <coproc>, #<opc1>, <CRd>, <CRn>, <CRm> {,#<opc2>}
```

where:

2 — If specified, selects encoding T2 / A2. If omitted, selects encoding T1 / A1.

<c><q> — See *Standard assembler syntax fields* on page A8-7. An ARM CDP2 instruction must be unconditional.

<coproc> — The name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, …, p15.

<opc1> — Is a coprocessor-specific opcode, in the range 0 to 15.

<CRd> — The destination coprocessor register for the instruction.

<CRn> — The coprocessor register that contains the first operand.

<CRm> — The coprocessor register that contains the second operand.

<opc2> — Is a coprocessor-specific opcode in the range 0 to 7. If it is omitted, <opc2> is 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_InternalOperation(cp, ThisInstr());
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
CDP{2}<c><q>  <coproc>, #<opc1>, <CRd>, <CRn>, <CRm> {,#<opc2>}
```

各項目の説明については以下を参照して下さい。

2 — 指定した場合、エンコード T2/A2 が選択されます。省略した場合、エンコード T1/A1 が選択されます。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM CDP2 は常に無条件命令です。

<coproc> — コプロセッサの名前。対応するコプロセッサ番号が命令の cp_num フィールドに配置されます。標準の汎用コプロセッサ名は、p0、p1、…、p15 です。

<opc1> — コプロセッサ固有のオペコードで、範囲は 0 ～ 15 です。

<CRd> — 命令のデスティネーションコプロセッサ レジスタ

<CRn> — 最初のオペランドを格納しているコプロセッサレジスタ

<CRm> — 2 番目のオペランドを格納しているコプロセッサレジスタ

<opc2> — コプロセッサ固有のオペコードで、範囲は 0 ～ 7 です。省略した場合、<opc2> が 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_InternalOperation(cp, ThisInstr());
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.29 CHKA

CHKA is a ThumbEE instruction. For details see *CHKA* on page A9-15.

A8.6.30 CLREX

Clear-Exclusive clears the local record of the executing processor that an address has had a request for an exclusive access.

Encoding T1 ARMv7

CLREX<c>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	0	1	0	(1)	(1)	(1)	(1)

```
// No additional decoding required
```

Encoding A1 ARMv6K, ARMv7

CLREX

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	(1)	(1)	(1)	(1)

```
// No additional decoding required
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.29 CHKA

CHKA は ThumbEE 命令です。詳細については、*CHKA* P. A9- 15 を参照して下さい。

A8.6.30 CLREX

CLREX（排他クリア）は、特定のアドレスに対して排他アクセスを要求した実行中のプロセッサのローカルレコードをクリアします。

エンコード T1 ARMv7

CLREX<c>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	0	1	0	(1)	(1)	(1)	(1)

```
// No additional decoding required
```

エンコード A1 ARMv6K、ARMv7

CLREX

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	(1)	(1)	(1)	(1)

```
// No additional decoding required
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
CLREX<c><q>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7. An ARM CLREX instruction must be unconditional.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
CLREX<c><q>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM CLREX は常に無条件命令です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.31 CLZ

Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value.

Encoding T1 ARMv6T2, ARMv7

```
CLZ<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	1	Rm				1	1	1	1	Rd				1	0	0	0	Rm			

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv5T*, ARMv6*, ARMv7

```
CLZ<c> <Rd>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	1	1	0	(1)	(1)	(1)	(1)	Rd				(1)	(1)	(1)	(1)	0	0	0	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.31 CLZ

CLZ（先行ゼロカウント）は、値の中で、最初に存在するバイナリのビット 1 より前にあるバイナリのビット 0 の数を返します。

エンコード T1 ARMv6T2、ARMv7

```
CLZ<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	1	Rm				1	1	1	1	Rd				1	0	0	0	Rm			

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv5T*、ARMv6*、ARMv7

```
CLZ<c> <Rd>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	1	1	0	(1)	(1)	(1)	(1)	Rd				(1)	(1)	(1)	(1)	0	0	0	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
CLZ<c><q>  <Rd>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rm> The register that contains the operand. Its number must be encoded twice in encoding T1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = CountLeadingZeroBits(R[m]);
    R[d] = result<31:0>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
CLZ<c><q>  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rm> オペランドを含むレジスタ。エンコード T1 では、Rm の番号を 2 回エンコードする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = CountLeadingZeroBits(R[m]);
    R[d] = result<31:0>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.32 CMN (immediate)

Compare Negative (immediate) adds a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

Encoding T1 ARMv6T2, ARMv7

```
CMN<c> <Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);  imm32 = ThumbExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
CMN<c> <Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	1	1	1	Rn				(0)	(0)	(0)	(0)	imm12											

```
n = UInt(Rn);  imm32 = ARMExpandImm(imm12);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.32 CMN（イミディエート）

CMN（比較否定（イミディエート））は、レジスタの値とイミディエート値を加算します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv6T2、ARMv7

```
CMN<c> <Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);  imm32 = ThumbExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
CMN<c> <Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	1	1	1	Rn				(0)	(0)	(0)	(0)	imm12											

```
n = UInt(Rn);  imm32 = ARMExpandImm(imm12);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
CMN<c><q>  <Rn>, #<const>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rn>` The register that contains the operand. SP can be used in Thumb as well as in ARM.

`<const>` The immediate value to be added to the value obtained from `<Rn>`. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
CMN<c><q>  <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rn>` オペランドを含むレジスタ。SP は、Thumb でも ARM でも指定できます。

`<const>` `<Rn>` から取得された値に加算されるイミディエート値。値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.33 CMN (register)

Compare Negative (register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
CMN<c> <Rn>,<Rm>
```

15 14 13 12 11 10	9 8 7 6	5 4 3	2 1 0
0 1 0 0 0 0	1 0 1 1	Rm	Rn

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
CMN<c>.W  <Rn>,<Rm>{,<shift>}
```

15 14 13 12 11 10	9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	1 0 0 0	1	Rn	(0)	imm3	1 1 1 1	imm2	type	Rm

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
CMN<c> <Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 0	0	1 0 1 1	1	Rn	(0) (0) (0) (0)	imm5	type	0	Rm

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.33 CMN（レジスタ）

CMN（比較否定（レジスタ））は、レジスタの値と、オプションでシフトされるレジスタの値を加算します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
CMN<c> <Rn>,<Rm>
```

15 14 13 12 11 10	9 8 7 6	5 4 3	2 1 0
0 1 0 0 0 0	1 0 1 1	Rm	Rn

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
CMN<c>.W  <Rn>,<Rm>{,<shift>}
```

15 14 13 12 11 10	9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	1 0 0 0	1	Rn	(0)	imm3	1 1 1 1	imm2	type	Rm

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
CMN<c> <Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 0	0	1 0 1 1	1	Rn	(0) (0) (0) (0)	imm5	type	0	Rm

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
CMN<c><q>  <Rn>, <Rm> {,<shift>}
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The first operand register. SP can be used in Thumb (encoding T2) as well as in ARM.

<Rm> The register that is optionally shifted and used as the second operand.

<shift> The shift to apply to the value read from <Rm>. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
CMN<c><q>  <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> 最初のオペランドレジスタ。SP は、Thumb（エンコード T2）でも ARM でも指定できます。

<Rm> オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ

<shift> <Rm> から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.34 CMN (register-shifted register)

Compare Negative (register-shifted register) adds a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
CMN<c> <Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 0 1 1	1	Rn	(0) (0) (0) (0)	Rs	0	type	1	Rm

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(type);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.34 CMN（レジスタによりシフトしたレジスタ）

CMN（比較否定（レジスタによりシフトしたレジスタ））は、レジスタの値とレジスタによりシフトしたレジスタの値とを加算します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
CMN<c> <Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 0 1 1	1	Rn	(0) (0) (0) (0)	Rs	0	type	1	Rm

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(type);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
CMN<c><q>  <Rn>, <Rm>, <type>  <Rs>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rn>` The first operand register.

`<Rm>` The register that is shifted and used as the second operand.

`<type>` The type of shift to apply to the value read from `<Rm>`. It must be one of:

- `ASR` Arithmetic shift right, encoded as type = 0b10
- `LSL` Logical shift left, encoded as type = 0b00
- `LSR` Logical shift right, encoded as type = 0b01
- `ROR` Rotate right, encoded as type = 0b11.

`<Rs>` The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
CMN<c><q>  <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rn>` 最初のオペランドレジスタ

`<Rm>` シフトされ、2 番目のオペランドとして使用されるレジスタ

`<type>` `<Rm>` から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- `ASR` 算術右シフト。type = 0b10 としてエンコードされます。
- `LSL` 論理左シフト。type = 0b00 としてエンコードされます。
- `LSR` 論理右シフト。type = 0b01 としてエンコードされます。
- `ROR` 右ローテート。type = 0b11 としてエンコードされます。

`<Rs>` 最下位バイトにシフトのビット数を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.35 CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
CMP<c> <Rn>,#<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Rn			imm8							

```
n = UInt(Rdn);  imm32 = ZeroExtend(imm8, 32);
```

Encoding T2 ARMv6T2, ARMv7

```
CMP<c>.W <Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);  imm32 = ThumbExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
CMP<c> <Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	1	0	1	Rn				(0)	(0)	(0)	(0)	imm12											

```
n = UInt(Rn);  imm32 = ARMExpandImm(imm12);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.35 CMP（イミディエート）

CMP（比較（イミディエート））は、レジスタの値からイミディエート値を減算します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
CMP<c> <Rn>,#<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Rn			imm8							

```
n = UInt(Rdn);  imm32 = ZeroExtend(imm8, 32);
```

エンコード T2 ARMv6T2、ARMv7

```
CMP<c>.W <Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);  imm32 = ThumbExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
CMP<c> <Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	1	0	1	Rn				(0)	(0)	(0)	(0)	imm12											

```
n = UInt(Rn);  imm32 = ARMExpandImm(imm12);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
CMP<c><q>  <Rn>, #<const>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The first operand register. SP can be used in Thumb (encoding T2) as well as in ARM.

<const> The immediate value to be compared with the value obtained from <Rn>. The range of values is 0-255 for encoding T1. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values for encoding T2 and A1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
CMP<c><q>  <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> 最初のオペランドレジスタ。SP は、Thumb（エンコード T2）でも ARM でも指定できます。

<const> <Rn> から取得された値と比較されるイミディエート値。値の範囲は、エンコード T1 では 0 ～ 255 です。エンコード T2 および A1 での値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.36 CMP (register)

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

`CMP<c> <Rn>,<Rm>` `<Rn>` and `<Rm>` both from R0-R7

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	0	Rm			Rn		

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv4T, ARMv5T*, ARMv6*, ARMv7

`CMP<c> <Rn>,<Rm>` `<Rn>` and `<Rm>` not both from R0-R7

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	N	Rm				Rn		

```
n = UInt(N:Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
if n < 8 && m < 8 then UNPREDICTABLE;
if n == 15 || m == 15 then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

`CMP<c>.W <Rn>, <Rm> {,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	0	1	1	Rn				(0)	imm3			1	1	1	1	imm2		type		Rm			

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

`CMP<c> <Rn>,<Rm>{,<shift>}`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	1	0	1	Rn				(0)	(0)	(0)	(0)	imm5					type		0	Rm			

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.36 CMP（レジスタ）

CMP（比較（レジスタ））は、レジスタの値から、オプションとしてシフトされたレジスタの値を減算します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

`CMP<c> <Rn>,<Rm>` `<Rn>` および `<Rm>` の両方が R0 ～ R7

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	0	Rm			Rn		

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

`CMP<c> <Rn>,<Rm>` `<Rn>` および `<Rm>` の両方が R0 ～ R7 ではない

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	N	Rm				Rn		

```
n = UInt(N:Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
if n < 8 && m < 8 then UNPREDICTABLE;
if n == 15 || m == 15 then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

`CMP<c>.W <Rn>, <Rm> {,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	0	1	1	Rn				(0)	imm3			1	1	1	1	imm2		type		Rm			

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

`CMP<c> <Rn>,<Rm>{,<shift>}`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	1	0	1	Rn				(0)	(0)	(0)	(0)	imm5					type		0	Rm			

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
CMP<c><q>  <Rn>, <Rm> {,<shift>}
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The first operand register. The SP can be used.

<Rm> The register that is optionally shifted and used as the second operand. This register can be SP in both ARM and Thumb instructions, but:

- the use of SP is deprecated
- when assembling for the Thumb instruction set, only encoding T2 is available.

<shift> The shift to apply to the value read from <Rm>. If present, encodings T1 and T2 are not permitted. If absent, no shift is applied and all encodings are permitted. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
CMP<c><q>  <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> 最初のオペランドレジスタ。このレジスタには SP を指定できます。

<Rm> オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ。ARM 命令と Thumb 命令の両方で、このレジスタに SP を指定できますが、次の条件が適用されます。

- SP の使用は推奨されません。
- Thumb 命令セットのアセンブル時は、エンコード T2 のみ使用できます。

<shift> <Rm> から読み出された値に適用されるシフト。指定した場合、エンコード T1 および T2 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.37 CMP (register-shifted register)

Compare (register-shifted register) subtracts a register-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
CMP<c> <Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 0 1 0	1	Rn	(0) (0) (0) (0)	Rs	0	type	1	Rm

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(type);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.37 CMP（レジスタによりシフトしたレジスタ）

CMP（比較（レジスタによりシフトしたレジスタ））は、レジスタの値から、レジスタによりシフトしたレジスタの値を減算します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
CMP<c> <Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 0 1 0	1	Rn	(0) (0) (0) (0)	Rs	0	type	1	Rm

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(type);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
CMP<c><q>  <Rn>, <Rm>, <type>  <Rs>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The first operand register.

<Rm> The register that is shifted and used as the second operand.

<type> The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR Arithmetic shift right, encoded as type = 0b10
- LSL Logical shift left, encoded as type = 0b00
- LSR Logical shift right, encoded as type = 0b01
- ROR Rotate right, encoded as type = 0b11.

<Rs> The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
CMP<c><q>  <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> 最初のオペランドレジスタ

<Rm> シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR 算術右シフト。type = 0b10 としてエンコードされます。
- LSL 論理左シフト。type = 0b00 としてエンコードされます。
- LSR 論理右シフト。type = 0b01 としてエンコードされます。
- ROR 右ローテート。type = 0b11 としてエンコードされます。

<Rs> 最下位バイトにシフトのビット数を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.38 CPS

Change Processor State is a system instruction. For details see *CPS* on page B6-3.

A8.6.39 CPY

Copy is a pre-UAL synonym for `MOV` (register).

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.38 CPS

CPS（プロセッサ状態変更）はシステム命令です。詳細については、*CPS* P. B6-3 を参照して下さい。

A8.6.39 CPY

CPY（コピー）は、UAL 以前の `MOV`（レジスタ）と同義語です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
CPY  <Rd>, <Rn>
```

This is equivalent to:

```
MOV  <Rd>, <Rn>
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
CPY  <Rd>, <Rn>
```

これは、次の命令と等価です。

```
MOV  <Rd>, <Rn>
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.40 DBG

Debug Hint provides a hint to debug and related systems. See their documentation for what use (if any) they make of this instruction.

Encoding T1 ARMv7 (executes as NOP in ARMv6T2)

```
DBG<c> #<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	1	1	1	1	option			

```
// Any decoding of 'option' is specified by the debug system
```

Encoding A1 ARMv7 (executes as NOP in ARMv6Kand ARMv6T2)

```
DBG<c> #<option>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	1	1	1	1	option			

```
// Any decoding of 'option' is specified by the debug system
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.40 DBG

DBG（デバッグヒント）は、デバッグおよび関連するシステムに対してヒントを提供します。この命令の使用方法については、該当する説明書を参照して下さい。

エンコード T1 ARMv7（ARMv6T2 では NOP として動作する）

```
DBG<c> #<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	1	1	1	1	option			

```
// Any decoding of 'option' is specified by the debug system
```

エンコード A1 ARMv7（ARMv6K と ARMv6T2 では NOP として動作する）

```
DBG<c> #<option>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	1	1	1	1	option			

```
// Any decoding of 'option' is specified by the debug system
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
DBG<c><q>  #<option>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<option> Provides extra information about the hint, and is in the range 0 to 15.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Debug(option);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
DBG<c><q>  #<option>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<option> ヒントに関するその他の情報を提供します。範囲は 0 ～ 15 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Debug(option);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.41 DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see *Data Memory Barrier (DMB)* on page A3-48.

Encoding T1 ARMv7

```
DMB<c> #<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	1	option			

```
// No additional decoding required
```

Encoding A1 ARMv7

```
DMB #<option>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	1	option			

```
// No additional decoding required
```

Assembler syntax

```
DMB<c><q>  {<opt>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `DMB` instruction must be unconditional.

`<opt>` Specifies an optional limitation on the DMB operation. Values are:

- `SY` Full system is the required shareability domain, reads and writes are the required access types. Can be omitted.
  This option is referred to as the full system DMB. Encoded as option == '1111'.
- `ST` Full system is the required shareability domain, writes are the required access type. `SYST` is a synonym for `ST`. Encoded as option == '1110'.
- `ISH` Inner Shareable is the required shareability domain, reads and writes are the required access types. Encoded as option == '1011'.
- `ISHST` Inner Shareable is the required shareability domain, writes are the required access type. Encoded as option == '1010'.
- `NSH` Non-shareable is the required shareability domain, reads and writes are the required access types. Encoded as option == '0111'.
- `NSHST` Non-shareable is the required shareability domain, writes are the required access type. Encoded as option == '0110'.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.41 DMB

DMB（データ メモリバリア）は、メモリアクセスの観測の順序を保証するメモリバリアです。詳細については、*データ メモリバリア (DMB)* P. A3-48 を参照して下さい。

エンコード T1 ARMv7

```
DMB<c> #<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	1	option			

```
// No additional decoding required
```

エンコード A1 ARMv7

```
DMB #<option>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	1	option			

```
// No additional decoding required
```

アセンブラ構文

```
DMB<c><q>  {<opt>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `DMB` は常に無条件命令です。

`<opt>` オプションで、DMB 動作に対する制限を指定します。値は次のとおりです。

- `SY` 完全なシステムが共有可必須ドメインであり、読み出しおよび書き込みが必要アクセスタイプです。省略することができます。
  このオプションは、完全なシステム DMB と呼ばれます。option == '1111' としてエンコードされます。
- `ST` 完全なシステムが共有可必須ドメインであり、書き込みが必要アクセスタイプです。`SYST` は、`ST` と同義語です。option == '1110' としてエンコードされます。
- `ISH` 内部共有可が共有可必須ドメインであり、読み出しおよび書き込みが必要アクセスタイプです。option == '1011' としてエンコードされます。
- `ISHST` 内部共有可が共有可必須ドメインであり、書き込みが必要アクセスタイプです。option == '1010' としてエンコードされます。
- `NSH` 共有不可が共有可必須ドメインであり、読み出しおよび書き込みが必要アクセスタイプです。option == '0111' としてエンコードされます。
- `NSHST` 共有不可が共有可必須ドメインであり、書き込みが必要アクセスタイプです。option == '0110' としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

OSH Outer Shareable is the required shareability domain, reads and writes are the required access types. Encoded as option == '0011'.

OSHST Outer Shareable is the required shareability domain, writes are the required access type. Encoded as option == '0010'.

All other encodings of option are reserved. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system DMB operation, but software must not must rely on this operation.

Note

The following alternative <opt> values are supported, but ARM recommends that you do not use these alternative values:

- SH as an alias for ISH
- SHST as an alias for ISHST
- UN as an alias for NSH
- UNST is an alias for NSHST.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    case option of
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
        when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
        when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
        when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
        when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
        when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
        otherwise    domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;
    DataMemoryBarrier(domain, types);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

OSH 外部共有可が共有可必須ドメインであり、読み出しおよび書き込みが必要アクセスタイプです。option == '0011' としてエンコードされます。

OSHST 外部共有可が共有可必須ドメインであり、書き込みが必要アクセスタイプです。option == '0010' としてエンコードされます。

オプションの他のエンコードはすべて予約されています。SY 以外のオプションが実装されるかどうかは実装定義です。未サポートのオプションと予約されているオプションはすべて、完全なシステム DMB 動作として実行する必要がありますが、ソフトウェアはこの動作に依存することはできません。

注

代替の <opt> として次の値がサポートされていますが、これらの代替値の使用は推奨されません。

- ISH のエイリアスとしての SH
- ISHST のエイリアスとしての SHST
- NSH のエイリアスとしての UN
- NSHST のエイリアスとしての UNST

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    case option of
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
        when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
        when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
        when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
        when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
        when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
        otherwise    domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;
    DataMemoryBarrier(domain, types);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.42 DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier (DSB)* on page A3-49.

Encoding T1 ARMv7

```
DSB<c> #<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	option			

```
// No additional decoding required
```

Encoding A1 ARMv7

```
DSB #<option>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	0	option			

```
// No additional decoding required
```

Assembler syntax

```
DSB<c><q>  {<opt>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `DSB` instruction must be unconditional.

`<opt>` Specifies an optional limitation on the DSB operation. Values are:

- `SY` Full system is the required shareability domain, reads and writes are the required access types. Can be omitted.
  This option is referred to as the full system DMB. Encoded as option == '1111'.
- `ST` Full system is the required shareability domain, writes are the required access type. `SYST` is a synonym for `ST`. Encoded as option == '1110'.
- `ISH` Inner Shareable is the required shareability domain, reads and writes are the required access types. Encoded as option == '1011'.
- `ISHST` Inner Shareable is the required shareability domain, writes are the required access type. Encoded as option == '1010'.
- `NSH` Non-shareable is the required shareability domain, reads and writes are the required access types. Encoded as option == '0111'.
- `NSHST` Non-shareable is the required shareability domain, writes are the required access type. Encoded as option == '0110'.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.42 DSB

DSB（データ同期バリア）は、メモリアクセスの完了を保証するメモリバリアです。詳細については、*データ同期バリア (DSB)* P. A3- 49 を参照して下さい。

エンコード T1 ARMv7

```
DSB<c> #<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	option			

```
// No additional decoding required
```

エンコード A1 ARMv7

```
DSB #<option>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	0	option			

```
// No additional decoding required
```

アセンブラ構文

```
DSB<c><q>  {<opt>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `DSB` は常に無条件命令です。

`<opt>` オプションで、DSB 動作に対する制限を指定します。値は次のとおりです。

- `SY` 完全なシステムが共有可必須ドメインであり、読み出しおよび書き込みが必要アクセスタイプです。省略することができます。
  このオプションは、完全なシステム DMB と呼ばれます。option == '1111' としてエンコードされます。
- `ST` 完全なシステムが共有可必須ドメインであり、書き込みが必要アクセスタイプです。`SYST` は、`ST` と同義語です。option == '1110' としてエンコードされます。
- `ISH` 内部共有可が共有可必須ドメインであり、読み出しおよび書き込みが必要アクセスタイプです。option == '1011' としてエンコードされます。
- `ISHST` 内部共有可が共有可必須ドメインであり、書き込みが必要アクセスタイプです。option == '1010' としてエンコードされます。
- `NSH` 共有不可が共有可必須ドメインであり、読み出しおよび書き込みが必要アクセスタイプです。option == '0111' としてエンコードされます。
- `NSHST` 共有不可が共有可必須ドメインであり、書き込みが必要アクセスタイプです。option == '0110' としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

OSH Outer Shareable is the required shareability domain, reads and writes are the required access types. Encoded as option == '0011'.

OSHST Outer Shareable is the required shareability domain, writes are the required access type. Encoded as option == '0010'.

All other encodings of option are reserved. It is IMPLEMENTATION DEFINED whether options other than SY are implemented. All unsupported and reserved options must execute as a full system DSB operation, but software must not must rely on this operation.

Note

The following alternative <opt> values are supported, but ARM recommends that you do not use these alternative values:

- SH as an alias for ISH
- SHST as an alias for ISHST
- UN as an alias for NSH
- UNST is an alias for NSHST.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    case option of
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
        when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
        when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
        when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
        when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
        when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
        otherwise    domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;
    DataSynchronizationBarrier(domain, types);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

OSH 外部共有可が共有可必須ドメインであり、読み出しおよび書き込みが必要アクセスタイプです。option == '0011' としてエンコードされます。

OSHST 外部共有可が共有可必須ドメインであり、書き込みが必要アクセスタイプです。option == '0010' としてエンコードされます。

オプションの他のエンコードはすべて予約されています。SY 以外のオプションが実装されるかどうかは実装定義です。未サポートのオプションと予約されているオプションはすべて、完全なシステム DSB 動作として実行する必要がありますが、ソフトウェアはこの動作に依存することはできません。

注

代替の <opt> として次の値がサポートされていますが、これらの代替値の使用は推奨されません。

- ISH のエイリアスとしての SH
- ISHST のエイリアスとしての SHST
- NSH のエイリアスとしての UN
- NSHST のエイリアスとしての UNST

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    case option of
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
        when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
        when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
        when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
        when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
        when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
        otherwise    domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;
    DataSynchronizationBarrier(domain, types);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.43 ENTERX

ENTERX causes a change from Thumb state to ThumbEE state, or has no effect in ThumbEE state. For details see *ENTERX, LEAVEX* on page A9-7.

A8.6.44 EOR (immediate)

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv6T2, ARMv7

```
EOR{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	1	0	0	S	Rn				0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE TEQ (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
EOR{S}<c> <Rd>,<Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	0	0	1	S	Rn				Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.43 ENTERX

ENTERX は Thumb 状態から ThumbEE 状態への変更を引き起こしますが、ThumbEE 状態では何の影響もありません。詳細については、*ENTERX*、*LEAVEX* P. A9-7 を参照して下さい。

A8.6.44 EOR（イミディエート）

EOR（ビット単位排他的論理和（イミディエート））は、レジスタの値とイミディエート値との間でビット単位の排他的論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

```
EOR{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	1	0	0	S	Rn				0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE TEQ (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
EOR{S}<c> <Rd>,<Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	0	0	1	S	Rn				Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
EOR{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The register that contains the operand.
<const>	The immediate value to be exclusive ORed with the value obtained from <Rn>. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

The pre-UAL syntax EOR<c>S is equivalent to EORS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
EOR{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	オペランドを含むレジスタ
<const>	<Rn> から取得された値との排他的論理和が実行されるイミディエート値。値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

UAL 以前の構文の EOR<c>S は、EORS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.45 EOR (register)

Bitwise Exclusive OR (register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
EORS <Rdn>,<Rm>                    Outside IT block.
EOR<c> <Rdn>,<Rm>                  Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
EOR{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	1	0	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
if Rd == '1111' && S == '1' then SEE TEQ (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
EOR{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	0	0	0	1	S	Rn			Rd				imm5					type		0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.45 EOR（レジスタ）

EOR（ビット単位排他的論理和（レジスタ））は、レジスタの値と、オプションとしてシフトしたレジスタの値との間でビット単位の排他的論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
EORS <Rdn>,<Rm>                    IT ブロック外部
EOR<c> <Rdn>,<Rm>                  IT ブロック内部
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
EOR{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	1	0	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
if Rd == '1111' && S == '1' then SEE TEQ (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
EOR{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	0	0	0	1	S	Rn			Rd				imm5					type		0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
EOR{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The register that is optionally shifted and used as the second operand.

<shift> — The shift to apply to the value read from <Rm>. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted.*Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

In Thumb assembly:

- outside an IT block, if EORS <Rd>,<Rn>,<Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though EORS <Rd>,<Rn> had been written
- inside an IT block, if EOR<c> <Rd>,<Rn>,<Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though EOR<c> <Rd>,<Rn> had been written.

To prevent either of these happening, use the .W qualifier.

The pre-UAL syntax EOR<c>S is equivalent to EORS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
EOR{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ

<shift> — <Rm> から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

Thumb アセンブラでは、次の規則が適用されます。

- IT ブロック外部で、EORS <Rd>,<Rn>,<Rd> の <Rd> と <Rn> の両方が R0 ～ R7 の範囲に含まれる場合、EORS <Rd>,<Rn> と記述されたものとして、エンコード T1 を使用してアセンブルされます。
- IT ブロック内部で、EOR<c> <Rd>,<Rn>,<Rd> の <Rd> と <Rn> の両方が R0 ～ R7 の範囲に含まれる場合、EOR<c> <Rd>,<Rn> と記述されたものとして、エンコード T1 を使用してアセンブルされます。

これらの事態を避けるには、.W 修飾子を使用します。

UAL 以前の構文の EOR<c>S は、EORS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.46 EOR (register-shifted register)

Bitwise Exclusive OR (register-shifted register) performs a bitwise Exclusive OR of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
EOR{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 0 0 1	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.46 EOR（レジスタによりシフトしたレジスタ）

EOR（ビット単位排他的論理和（レジスタによりシフトしたレジスタ））は、レジスタの値と、レジスタによりシフトしたレジスタの値との間で、ビット単位の排他的論理和を実行します。結果をデスティネーションレジスタに書き込み、オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
EOR{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 0 0 1	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
EOR{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The register that is shifted and used as the second operand.

<type> — The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR — Arithmetic shift right, encoded as type = 0b10
- LSL — Logical shift left, encoded as type = 0b00
- LSR — Logical shift right, encoded as type = 0b01
- ROR — Rotate right, encoded as type = 0b11.

<Rs> — The register whose bottom byte contains the amount to shift by.

The pre-UAL syntax EOR<c>S is equivalent to EORS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
EOR{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> — <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR — 算術右シフト。type = 0b10 としてエンコードされます。
- LSL — 論理左シフト。type = 0b00 としてエンコードされます。
- LSR — 論理右シフト。type = 0b01 としてエンコードされます。
- ROR — 右ローテート。type = 0b11 としてエンコードされます。

<Rs> — 最下位バイトにシフトのビット数を含むレジスタ

UAL 以前の構文の EOR<c>S は、EORS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.47 F* (former VFP instruction mnemonics)

Table A8-2 lists the UAL equivalents of pre-UAL VFP instruction mnemonics.

Table A8-2 VFP instruction mnemonics

Former ARM assembler mnemonic	UAL equivalent	See
FABSD, FABSS	VABS	*VABS* on page A8-532
FADDD, FADDS	VADD	*VADD (floating-point)* on page A8-538
FCMP, FCMPE, FCMPEZ, FCMPZ	VCMP{E}	*VCMP, VCMPE* on page A8-572
FCONSTD, FCONSTS	VMOV	*VMOV (immediate)* on page A8-640
FCPYD, FCPYS	VMOV	*VMOV (register)* on page A8-642
FCVTDS, FCVTSD	VCVT	*VCVT (between double-precision and single-precision)* on page A8-584
FDIVD, FDIVS	VDIV	*VDIV* on page A8-590
FLDD	VLDR	*VLDR* on page A8-628
FLDMD, FLDMS	VLDM, VPOP	*VLDM* on page A8-626. *VPOP* on page A8-694
FLDMX	FLDMX	*FLDMX, FSTMX* on page A8-101
FLDS	VLDR	*VLDR* on page A8-628
FMACD, FMACS	VMLA	*VMLA, VMLS (floating-point)* on page A8-636
FMDHR, FMDLR	VMOV	*VMOV (ARM core register to scalar)* on page A8-644
FMDRR	VMOV	*VMOV (between two ARM core registers and a doubleword extension register)* on page A8-652
FMRDH, FMRDL	VMOV	*VMOV (scalar to ARM core register)* on page A8-646
FMRRD	VMOV	*VMOV (between two ARM core registers and a doubleword extension register)* on page A8-652
FMRRS	VMOV	*VMOV (between two ARM core registers and two single-precision registers)* on page A8-650
FMRS	VMOV	*VMOV (between ARM core register and single-precision register)* on page A8-648
FMRX	VMRS	*VMRS* on page A8-658
FMSCD, FMSCS	VNMLS	*VNMLA, VNMLS, VNMUL* on page A8-674
FMSR	VMOV	*VMOV (between ARM core register and single-precision register)* on page A8-648
FMSRR	VMOV	*VMOV (between two ARM core registers and two single-precision registers)* on page A8-650
FMSTAT	VMRS	*VMRS* on page A8-658
FMULD, FMULS	VMUL	*VMUL (floating-point)* on page A8-664
FMXR	VMSR	*VMSR* on page A8-660

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.47 F*（以前の VFP 命令ニーモニック）

UAL 以前の VFP 命令ニーモニックと等価な UAL のニーモニックの一覧を、表 A8-2 に示します。

表 A8-2 VFP 命令ニーモニック

以前の ARM アセンブラニーモニック	UAL での等価なニーモニック	参照先
FABSD、FABSS	VABS	*VABS* P. A8-532
FADDD、FADDS	VADD	*VADD（浮動小数点）* P. A8-538
FCMP、FCMPE、FCMPEZ、FCMPZ	VCMP{E}	*VCMP*、*VCMPE* P. A8-572
FCONSTD、FCONSTS	VMOV	*VMOV（イミディエート）* P. A8-640
FCPYD、FCPYS	VMOV	*VMOV（レジスタ）* P. A8-642
FCVTDS、FCVTSD	VCVT	*VCVT（倍精度と単精度との間）* P. A8-584
FDIVD、FDIVS	VDIV	*VDIV* P. A8-590
FLDD	VLDR	*VLDR* P. A8-628
FLDMD、FLDMS	VLDM、VPOP	*VLDM* P. A8-626. *VPOP* P. A8-694
FLDMX	FLDMX	*FLDMX*、*FSTMX* P. A8-101
FLDS	VLDR	*VLDR* P. A8-628
FMACD、FMACS	VMLA	*VMLA*、*VMLS（浮動小数点数）* P. A8-636
FMDHR、FMDLR	VMOV	*VMOV（ARM コアレジスタからスカラへのコピー）* P. A8-644
FMDRR	VMOV	*VMOV（2 つの ARM コアレジスタとダブルワード拡張機能レジスタとの間のコピー）* P. A8-652
FMRDH、FMRDL	VMOV	*VMOV（スカラから ARM コアレジスタへのコピー）* P. A8-646
FMRRD	VMOV	*VMOV（2 つの ARM コアレジスタとダブルワード拡張機能レジスタとの間のコピー）* P. A8-652
FMRRS	VMOV	*VMOV（2 つの ARM コアレジスタと 2 つの単精度レジスタとの間の転送）* P. A8-650
FMRS	VMOV	*VMOV（ARM コアレジスタと単精度レジスタとの間の転送）* P. A8-648
FMRX	VMRS	*VMRS* P. A8-658
FMSCD、FMSCS	VNMLS	*VNMLA*、*VNMLS*、*VNMUL* P. A8-674
FMSR	VMOV	*VMOV（ARM コアレジスタと単精度レジスタとの間の転送）* P. A8-648
FMSRR	VMOV	*VMOV（2 つの ARM コアレジスタと 2 つの単精度レジスタとの間の転送）* P. A8-650
FMSTAT	VMRS	*VMRS* P. A8-658
FMULD、FMULS	VMUL	*VMUL*、*VMULL（浮動小数点数）* P. A8-664
FMXR	VMSR	*VMSR* P. A8-660

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A8-2 VFP instruction mnemonics (continued)

Former ARM assembler mnemonic	UAL equivalent	See
FNEGD, FNEGS	VNEG	*VNEG* on page A8-672
FNMACD, FNMACS	VMLS	*VMLA, VMLS (floating-point)* on page A8-636
FNMSCD, FNMSCS	VNMLA	*VNMLA, VNMLS, VNMUL* on page A8-674
FNMULD, FNMULS	VNMUL	*VNMLA, VNMLS, VNMUL* on page A8-674
FSHTOD, FSHTOS	VCVT	*VCVT (between floating-point and fixed-point, VFP)* on page A8-582
FSITOD, FSITOS	VCVT	*VCVT, VCVTR (between floating-point and integer, VFP)* on page A8-578
FSLTOD, FSLTOS	VCVT	*VCVT (between floating-point and fixed-point, VFP)* on page A8-582
FSQRTD, FSQRTS	VSQRT	*VSQRT* on page A8-762
FSTD	VSTR	*VSTR* on page A8-786
FSTMD, FSTMS	VSTM, VPUSH	*VSTM* on page A8-784, *VPUSH* on page A8-696
FSTMX	FSTMX	*FLDMX, FSTMX*
FSTS	VSTR	*VSTR* on page A8-786
FSUBD, FSUBS	VSUB	*VSUB (floating-point)* on page A8-790
FTOSHD, FTOSHS	VCVT	*VCVT (between floating-point and fixed-point, VFP)* on page A8-582
FTOSI{Z}D, FTOSI{Z}S	VCVT{R}	*VCVT, VCVTR (between floating-point and integer, VFP)* on page A8-578
FTOSL, FTOUH	VCVT	*VCVT (between floating-point and fixed-point, VFP)* on page A8-582
FTOUI{Z}D, FTOUI{Z}S	VCVT{R}	*VCVT, VCVTR (between floating-point and integer, VFP)* on page A8-578
FTOULD, FTOULS, FUHTOD, FUHTOS	VCVT	*VCVT (between floating-point and fixed-point, VFP)* on page A8-582
FUITOD, FUITOS	VCVT	*VCVT, VCVTR (between floating-point and integer, VFP)* on page A8-578
FULTOD, FULTOS	VCVT	*VCVT (between floating-point and fixed-point, VFP)* on page A8-582

FLDMX, FSTMX

Encodings T1/A1 of the VLDM, VPOP, VPUSH, and VSTM instructions contain an imm8 field that is set to twice the number of doubleword registers to be transferred. Use of these encodings with an odd value in imm8 is deprecated, and there is no UAL syntax for them.

The pre-UAL mnemonics FLDMX and FSTMX result in the same instructions as FLDMD (VLDM.64 or VPOP.64) and FSTMD (VSTM.64 or VPUSH.64) respectively, except that imm8 is equal to twice the number of doubleword registers plus one. Use of FLDMX and FSTMX is deprecated from ARMv6, except for disassembly purposes, and reassembly of disassembled code.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 A8-2 VFP 命令ニーモニック（続き）

以前の ARM アセンブラニーモニック	UAL での等価なニーモニック	参照先
FNEGD、FNEGS	VNEG	*VNEG* P. A8-672
FNMACD、FNMACS	VMLS	*VMLA*、*VMLS（浮動小数点数）* P. A8-636
FNMSCD、FNMSCS	VNMLA	*VNMLA*、*VNMLS*、*VNMUL* P. A8-674
FNMULD、FNMULS	VNMUL	*VNMLA*、*VNMLS*、*VNMUL* P. A8-674
FSHTOD、FSHTOS	VCVT	*VCVT（浮動小数点数と固定小数点数との間、VFP）* P. A8-582
FSITOD、FSITOS	VCVT	*VCVT*、*VCVTR（浮動小数点数と整数との間、VFP）* P. A8-578
FSLTOD、FSLTOS	VCVT	*VCVT（浮動小数点数と固定小数点数との間、VFP）* P. A8-582
FSQRTD、FSQRTS	VSQRT	*VSQRT* P. A8-762
FSTD	VSTR	*VSTR* P. A8-786
FSTMD、FSTMS	VSTM、VPUSH	*VSTM* P. A8-784, *VPUSH* P. A8-696
FSTMX	FSTMX	*FLDMX*、*FSTMX*
FSTS	VSTR	*VSTR* P. A8-786
FSUBD、FSUBS	VSUB	*VSUB（浮動小数点数）* P. A8-790
FTOSHD、FTOSHS	VCVT	*VCVT（浮動小数点数と固定小数点数との間、VFP）* P. A8-582
FTOSI{Z}D、FTOSI{Z}S	VCVT{R}	*VCVT*、*VCVTR（浮動小数点数と整数との間、VFP）* P. A8-578
FTOSL、FTOUH	VCVT	*VCVT（浮動小数点数と固定小数点数との間、VFP）* P. A8-582
FTOUI{Z}D、FTOUI{Z}S	VCVT{R}	*VCVT*、*VCVTR（浮動小数点数と整数との間、VFP）* P. A8-578
FTOULD、FTOULS、FUHTOD、FUHTOS	VCVT	*VCVT（浮動小数点数と固定小数点数との間、VFP）* P. A8-582
FUITOD、FUITOS	VCVT	*VCVT*、*VCVTR（浮動小数点数と整数との間、VFP）* P. A8-578
FULTOD、FULTOS	VCVT	*VCVT（浮動小数点数と固定小数点数との間、VFP）* P. A8-582

FLDMX、FSTMX

VLDM、VPOP、VPUSH、VSTM の各命令のエンコード T1/A1 には、転送されるダブルワードレジスタの数の 2 倍に設定されている imm8 フィールドが含まれています。これらのエンコードを imm8 の奇数値とともに使用することは推奨されません。また、これらのエンコード用の UAL 構文は存在しません。

UAL 以前のニーモニック FLDMX および FSTMX の結果はそれぞれ、FLDMD（VLDM.64 または VPOP.64）および FSTMD（VSTM.64 または VPUSH.64）と同じ命令になります。ただし、例外として、imm8 は、ダブルワードレジスタの数の 2 倍 + 1 に等しくなります。ARMv6 および以降では、逆アセンブリ目的と、逆アセンブルされたコードの再アセンブリを除き、FLDMX および FSTMX の使用は推奨されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.48 HB, HBL, HBLP, HBP

These are ThumbEE instructions. For details see *HB, HBL* on page A9-16, *HBLP* on page A9-17, and *HBP* on page A9-18.

A8.6.49 ISB

Instruction Synchronization Barrier flushes the pipeline in the processor, so that all instructions following the ISB are fetched from cache or memory, after the instruction has been completed. It ensures that the effects of context altering operations, such as changing the ASID, or completed TLB maintenance operations, or branch predictor maintenance operations, as well as all changes to the CP15 registers, executed before the ISB instruction are visible to the instructions fetched after the ISB.

In addition, any branches that appear in program order after the ISB instruction are written into the branch prediction logic with the context that is visible after the ISB instruction. This is needed to ensure correct execution of the instruction stream.

Encoding T1 ARMv7

```
ISB<c> #<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	0	option			

```
// No additional decoding required
```

Encoding A1 ARMv7

```
ISB #<option>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	1	0	option			

```
// No additional decoding required
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.48 HB、HBL、HBLP、HBP

これらは ThumbEE 命令です。詳細については、*HB*、*HBL* P. A9- 16、*HBLP* P. A9- 17、*HBP* P. A9- 18 を参照して下さい。

A8.6.49 ISB

ISB（命令同期バリア）は、プロセッサ内のパイプラインをフラッシュして、ISB 命令よりも後にあるすべての命令が、この命令の完了後にキャッシュまたはメモリからフェッチされるようにします。これによって、ISB 命令の前に実行された、ASID の変更、TLB 保守操作の完了、分岐予測器保守操作の完了などのコンテキスト変更動作、および CP15 レジスタに対するすべての変更の影響が、ISB 命令の後にフェッチされた命令から可視であることが保証されます。

さらに、プログラム順序で ISB 命令の後に出現するすべての分岐命令は、ISB 命令後に可視になるコンテキストとともに分岐予測ロジックに書き込まれます。これは、命令ストリームの正確な実行を保証するために必要になります。

エンコード T1 ARMv7

```
ISB<c> #<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	0	option			

```
// No additional decoding required
```

エンコード A1 ARMv7

```
ISB #<option>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	1	0	option			

```
// No additional decoding required
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ISB<c><q>  {<opt>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `ISB` instruction must be unconditional.

`<opt>` Specifies an optional limitation on the ISB operation. Values are:

`SY` Full system ISB operation, encoded as option == '1111'. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as full system ISB operations, but must not be relied upon by software.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier();
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ISB<c><q>  {<opt>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `ISB` は常に無条件命令です。

`<opt>` ISB 動作に対するオプションの制限を指定します。値は次のとおりです。

`SY` 完全なシステム ISB 動作を指定し、option == '1111' としてエンコードされます。省略可能。

オプションの他のエンコードはすべて予約されています。これらに対応する命令は完全なシステム ISB 動作として実行されますが、ソフトウェアではこの動作を前提にしないようにする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier();
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.50 IT

If Then makes up to four following instructions (the *IT block*) conditional. The conditions for the instructions in the IT block can be the same, or some of them can be the inverse of others.

IT does not affect the condition code flags. Branches to any instruction in the IT block are not permitted, apart from those performed by exception returns.

16-bit instructions in the IT block, other than CMP, CMN and TST, do not set the condition code flags. The AL condition can be specified to get this changed behavior without conditional execution.

See also *ITSTATE* on page A2-17, *Conditional instructions* on page A4-4, and *Conditional execution* on page A8-8.

Encoding T1 ARMv6T2, ARMv7

IT{x{y{z}}} <firstcond> Not permitted in IT block

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 0 1 1	1 1 1 1	firstcond	mask

```
if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' then UNPREDICTABLE;
if firstcond == '1110' && BitCount(mask) != 1 then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

Related encodings See *If-Then, and hints* on page A6-12

Assembler syntax

IT{x{y{z}}}<q> <firstcond>

where:

<x> The condition for the second instruction in the IT block.

<y> The condition for the third instruction in the IT block.

<z> The condition for the fourth instruction in the IT block.

<q> See *Standard assembler syntax fields* on page A8-7. An IT instruction must be unconditional.

<firstcond> The condition for the first instruction in the IT block. See Table A8-1 on page A8-8 for the range of conditions available, and the encodings.

Each of <x>, <y>, and <z> can be either:

T Then. The condition attached to the instruction is <firstcond>.

E Else. The condition attached to the instruction is the inverse of <firstcond>. The condition code is the same as <firstcond>, except that the least significant bit is inverted. E must not be specified if <firstcond> is AL.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.50 IT

IT（If Then）は、以降の最大 4 つの命令（*IT ブロック*）を条件付きにします。IT ブロック内部の命令の条件は、すべてを同じにすることも、一部を反転させることもできます。

IT は、条件コードフラグに影響を与えません。IT ブロック内部の任意の命令への分岐は、例外からの復帰で実行される場合以外は許可されません。

CMP、CMN、TST を除く IT ブロック内部の 16 ビット命令では、条件コードフラグがセットされません。AL 条件を指定すると、条件付き実行を行わず、動作だけをこのように変更することができます。

ITSTATE P. A2-17、*条件付き命令* P. A4-4、*条件付き実行* P. A8-8 も参照して下さい。

エンコード T1 ARMv6T2、ARMv7

IT{x{y{z}}} <firstcond> IT ブロック内部では許可されない

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 0 1 1	1 1 1 1	firstcond	mask

```
if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' then UNPREDICTABLE;
if firstcond == '1110' && BitCount(mask) != 1 then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

関連エンコード *If-Then とヒント* P. A6-12 を参照して下さい。

アセンブラ構文

IT{x{y{z}}}<q> <firstcond>

各項目の説明については以下を参照して下さい。

<x> IT ブロック内部の 2 番目の命令に対する条件。

<y> IT ブロック内部の 3 番目の命令に対する条件。

<z> IT ブロック内部の 4 番目の命令に対する条件。

<q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。IT は常に無条件命令です。

<firstcond> IT ブロック内部の最初の命令に対する条件。使用可能な条件の範囲とエンコードについては、P. A8-8 の表 A8-1 を参照して下さい。

<x>、<y>、<z> には次のいずれかを指定できます。

T Then。命令に付加される条件は <firstcond> です。

E Else。命令に付加される条件は <firstcond> の逆です。条件コードは、最下位ビットが反転していること以外、<firstcond> と同じです。<firstcond> が AL の場合は、E を指定できません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A8-3 shows how the values of <x>, <y>, and <z> determine the value of the mask field.

Table A8-3 Determination of mask[a] field

<x>	<y>	<z>	mask[3]	mask[2]	mask[1]	mask[0]
Omitted	Omitted	Omitted	1	0	0	0
T	Omitted	Omitted	firstcond[0]	1	0	0
E	Omitted	Omitted	NOT firstcond[0]	1	0	0
T	T	Omitted	firstcond[0]	firstcond[0]	1	0
E	T	Omitted	NOT firstcond[0]	firstcond[0]	1	0
T	E	Omitted	firstcond[0]	NOT firstcond[0]	1	0
E	E	Omitted	NOT firstcond[0]	NOT firstcond[0]	1	0
T	T	T	firstcond[0]	firstcond[0]	firstcond[0]	1
E	T	T	NOT firstcond[0]	firstcond[0]	firstcond[0]	1
T	E	T	firstcond[0]	NOT firstcond[0]	firstcond[0]	1
E	E	T	NOT firstcond[0]	NOT firstcond[0]	firstcond[0]	1
T	T	E	firstcond[0]	firstcond[0]	NOT firstcond[0]	1
E	T	E	NOT firstcond[0]	firstcond[0]	NOT firstcond[0]	1
T	E	E	firstcond[0]	NOT firstcond[0]	NOT firstcond[0]	1
E	E	E	NOT firstcond[0]	NOT firstcond[0]	NOT firstcond[0]	1

a. Note that at least one bit is always 1 in mask.

The conditions specified in an IT instruction must match those specified in the syntax of the instructions in its IT block. When assembling to ARM code, assemblers check IT instruction syntax for validity but do not generate assembled instructions for them. See *Conditional instructions* on page A4-4.

Operation

```
EncodingSpecificOperations();
ITSTATE.IT<7:0> = firstcond:mask;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A8-3 は、<x>、<y>、および <z> の値によって mask フィールドの値がどのように決定されるかを示しています。

表 A8-3 mask [a] フィールドの値

<x>	<y>	<z>	mask[3]	mask[2]	mask[1]	mask[0]
省略	省略	省略	1	0	0	0
T	省略	省略	firstcond[0]	1	0	0
E	省略	省略	NOT firstcond[0]	1	0	0
T	T	省略	firstcond[0]	firstcond[0]	1	0
E	T	省略	NOT firstcond[0]	firstcond[0]	1	0
T	E	省略	firstcond[0]	NOT firstcond[0]	1	0
E	E	省略	NOT firstcond[0]	NOT firstcond[0]	1	0
T	T	T	firstcond[0]	firstcond[0]	firstcond[0]	1
E	T	T	NOT firstcond[0]	firstcond[0]	firstcond[0]	1
T	E	T	firstcond[0]	NOT firstcond[0]	firstcond[0]	1
E	E	T	NOT firstcond[0]	NOT firstcond[0]	firstcond[0]	1
T	T	E	firstcond[0]	firstcond[0]	NOT firstcond[0]	1
E	T	E	NOT firstcond[0]	firstcond[0]	NOT firstcond[0]	1
T	E	E	firstcond[0]	NOT firstcond[0]	NOT firstcond[0]	1
E	E	E	NOT firstcond[0]	NOT firstcond[0]	NOT firstcond[0]	1

a. マスク内では常に 1 のビットが 1 つ以上存在する必要があることに注意して下さい。

IT 命令で指定された条件は、IT ブロック内部の命令の構文で指定された条件と一致する必要があります。ARM コードへアセンブルするとき、アセンブラは IT 命令の構文が有効であるかどうかチェックしますが、アセンブルされた命令は生成しません。詳細については、*条件付き命令* P. A4-4 を参照して下さい。

動作

```
EncodingSpecificOperations();
ITSTATE.IT<7:0> = firstcond:mask;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.51 LDC, LDC2 (immediate)

Load Coprocessor loads memory data from a sequence of consecutive memory addresses to a coprocessor. If no coprocessor can execute the instruction, an Undefined Instruction exception is generated.

This is a generic coprocessor instruction. Some of the fields have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer. These fields are the D bit, the CRd field, and in the Unindexed addressing mode only, the imm8 field.

For more information about the coprocessors see *Coprocessor support* on page A2-68.

Encoding T1 / A1 ARMv6T2, ARMv7 for encoding T1
ARMv4*, ARMv5T*, ARMv6*, ARMv7 for encoding A1

```
LDC{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm>]{!}
LDC{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm>
LDC{L}<c> <coproc>,<CRd>,[<Rn>],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	1	Rn				CRd				coproc				imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	P	U	D	W	1	Rn				CRd				coproc				imm8							

```
if Rn == '1111' then SEE LDC (literal);
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
if coproc == '101x' then SEE "Advanced SIMD and VFP";
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

Encoding T2 / A2 ARMv6T2, ARMv7 for encoding T2
ARMv5T*, ARMv6*, ARMv7 for encodingA2

```
LDC2{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm>]{!}
LDC2{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm>
LDC2{L}<c> <coproc>,<CRd>,[<Rn>],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	1	Rn				CRd				coproc				imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	1	Rn				CRd				coproc				imm8							

```
if Rn == '1111' then SEE LDC (literal);
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

Advanced SIMD and VFP See *Extension register load/store instructions* on page A7-26

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.51 LDC、LDC2（イミディエート）

LDC、LDC2（コプロセッサロード（イミディエート））は、連続するメモリアドレスからコプロセッサにメモリデータをロードします。コプロセッサが命令を実行できない場合は、未定義命令例外が発生します。

これは、汎用のコプロセッサ命令です。一部のフィールドの機能はアーキテクチャで定義されていないため、コプロセッサ命令セットの設計者が自由に使用することができます。これらのフィールドは、D ビット、CRd フィールド、imm8 フィールド（インデクスなしアドレシングモードのみ）です。

コプロセッサの詳細については、*コプロセッサのサポート* P. A2-68 を参照して下さい。

エンコード T1/A1 ARMv6T2、ARMv7（エンコード T1 の場合）
ARMv4*、ARMv5T*、ARMv6*、ARMv7（エンコード A1 の場合）

```
LDC{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm>]{!}
LDC{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm>
LDC{L}<c> <coproc>,<CRd>,[<Rn>],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	1	Rn				CRd				coproc				imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	P	U	D	W	1	Rn				CRd				coproc				imm8							

```
if Rn == '1111' then SEE LDC (literal);
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
if coproc == '101x' then SEE "Advanced SIMD and VFP";
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

エンコード T2/A2 ARMv6T2、ARMv7（エンコード T2 の場合）
ARMv5T*、ARMv6*、ARMv7（エンコード A2 の場合）

```
LDC2{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm>]{!}
LDC2{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm>
LDC2{L}<c> <coproc>,<CRd>,[<Rn>],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	1	Rn				CRd				coproc				imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	1	Rn				CRd				coproc				imm8							

```
if Rn == '1111' then SEE LDC (literal);
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

アドバンスト SIMD および VFP *拡張機能レジスタロード／ストア命令* P. A7-26 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>{,#+/-<imm>}]`	Offset. P = 1, W = 0.
`LDC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>,#+/-<imm>]!`	Pre-indexed. P = 1, W = 1.
`LDC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],#+/-<imm>`	Post-indexed. P = 0, W = 1.
`LDC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],<option>`	Unindexed. P = 0, W = 0, U = 1.

where:

`2` If specified, selects encoding T2 / A2. If omitted, selects encoding T1 / A1.

`L` If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `LDC2` instruction must be unconditional.

`<coproc>` The name of the coprocessor. The standard generic coprocessor names are p0, p1, …, p15.

`<CRd>` The coprocessor destination register.

`<Rn>` The base register. The SP can be used. For PC use see *LDC, LDC2 (literal)* on page A8-108.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. Values are multiples of 4 in the range 0-1020. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of +0.

`<option>` A coprocessor option. An integer in the range 0-255 enclosed in { }. Encoded in imm8.

The pre-UAL syntax `LDC<c>L` is equivalent to `LDCL<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        NullCheckIfThumbEE(n);
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        repeat
            Coproc_SendLoadedWord(MemA[address,4], cp, ThisInstr());  address = address + 4;
        until Coproc_DoneLoading(cp, ThisInstr());
        if wback then R[n] = offset_addr;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>{,#+/-<imm>}]`	オフセット：P = 1、W = 0
`LDC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>,#+/-<imm>]!`	プリインデクス：P = 1、W = 1
`LDC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],#+/-<imm>`	ポストインデクス：P = 0、W = 1
`LDC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],<option>`	インデクスなし：P = 0、W = 0、U = 1

各項目の説明については以下を参照して下さい。

`2` 指定した場合、エンコード T2/A2 が選択されます。省略した場合、エンコード T1/A1 が選択されます。

`L` 指定した場合、エンコードの D == 1 の形式が選択されます。省略した場合、D == 0 の形式が選択されます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。ARM `LDC2` は常に無条件命令です。

`<coproc>` コプロセッサの名前。標準の汎用コプロセッサ名は、p0、p1、…、p15 です。

`<CRd>` コプロセッサのデスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC の指定については、*LDC*、*LDC2（リテラル）* P. A8- 108 を参照して下さい。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。値は、0 ～ 1020 までの 4 の倍数です。オフセットアドレシング構文では、`<imm>` を省略して、+ 0 のオフセットを指定することができます。

`<option>` コプロセッサオプション。{} で囲んだ 0 ～ 255 の整数。imm8 にエンコードされます。

UAL 以前の構文の `LDC<c>L` は、`LDCL<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        NullCheckIfThumbEE(n);
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        repeat
            Coproc_SendLoadedWord(MemA[address,4], cp, ThisInstr());  address = address + 4;
        until Coproc_DoneLoading(cp, ThisInstr());
        if wback then R[n] = offset_addr;
```

例外

未定義命令、データアボート。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.52 LDC, LDC2 (literal)

Load Coprocessor loads memory data from a sequence of consecutive memory addresses to a coprocessor. If no coprocessor can execute the instruction, an Undefined Instruction exception is generated.

This is a generic coprocessor instruction. The D bit and the CRd field have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer.

For more information about the coprocessors see *Coprocessor support* on page A2-68.

Encoding T1 / A1 ARMv6T2, ARMv7 for encoding T1
ARMv4*, ARMv5T*, ARMv6*, ARMv7 for encoding A1

```
LDC{L}<c> <coproc>,<CRd>,<label>
LDC{L}<c> <coproc>,<CRd>,[PC,#-0]                    Special case
LDC{L}<c> <coproc>,<CRd>,[PC],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	1	1	1	1	1	CRd				coproc				imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	P	U	D	W	1	1	1	1	1	CRd				coproc				imm8							

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
if coproc == '101x' then SEE "Advanced SIMD and VFP";
index = (P == '1');  add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

Encoding T2 / A2 ARMv6T2, ARMv7 for encoding T2
ARMv5T*, ARMv6*, ARMv7 for encodingA2

```
LDC2{L}<c> <coproc>,<CRd>,<label>
LDC2{L}<c> <coproc>,<CRd>,[PC,#-0]                   Special case
LDC2{L}<c> <coproc>,<CRd>,[PC],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	1	1	1	1	1	CRd				coproc				imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	1	1	1	1	1	CRd				coproc				imm8							

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
index = (P == '1');  add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

Advanced SIMD and VFP See *Extension register load/store instructions* on page A7-26

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.52 LDC、LDC2（リテラル）

LDC、LDC2（コプロセッサロード（リテラル））は、連続するメモリアドレスからコプロセッサにメモリデータをロードします。コプロセッサが命令を実行できない場合は、未定義命令例外が発生します。

これは、汎用のコプロセッサ命令です。D ビットと CRd フィールドの機能はアーキテクチャで定義されていないため、コプロセッサ命令セットの設計者が自由に使用することができます。

コプロセッサの詳細については、*コプロセッサのサポート* P. A2-68 を参照して下さい。

エンコード T1/A1 ARMv6T2、ARMv7（エンコード T1 の場合）
ARMv4*、ARMv5T*、ARMv6*、ARMv7（エンコード A1 の場合）

```
LDC{L}<c> <coproc>,<CRd>,<label>
LDC{L}<c> <coproc>,<CRd>,[PC,#-0]                    特殊なケース
LDC{L}<c> <coproc>,<CRd>,[PC],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	1	1	1	1	1	CRd				coproc				imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	P	U	D	W	1	1	1	1	1	CRd				coproc				imm8							

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
if coproc == '101x' then SEE "Advanced SIMD and VFP";
index = (P == '1');  add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

エンコード T2/A2 ARMv6T2、ARMv7（エンコード T2 の場合）
ARMv5T*、ARMv6*、ARMv7（エンコード A2 の場合）

```
LDC2{L}<c> <coproc>,<CRd>,<label>
LDC2{L}<c> <coproc>,<CRd>,[PC,#-0]                   特殊なケース
LDC2{L}<c> <coproc>,<CRd>,[PC],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	1	1	1	1	1	CRd				coproc				imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	1	1	1	1	1	CRd				coproc				imm8							

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
index = (P == '1');  add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

アドバンスト SIMD および VFP *拡張機能レジスタロード／ストア命令* P. A7-26 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDC{2}{L}<c><q> <coproc>, <CRd>, <label>`	Normal form with P = 1, W = 0
`LDC{2}{L}<c><q> <coproc>, <CRd>, [PC,#+/-<imm>]`	Alternative form with P = 1, W = 0
`LDC{2}{L}<c><q> <coproc>, <CRd>, [PC], <option>`	Unindexed form with P = 0, U = 1, W = 0

where:

`2`	If specified, selects encoding T2 / A2. If omitted, selects encoding T1 / A1.
`L`	If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.
`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM `LDC2` instruction must be unconditional.
`<coproc>`	The name of the coprocessor. The standard generic coprocessor names are p0, p1, …, p15.
`<CRd>`	The coprocessor destination register.
`<label>`	The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC,4)` value of this instruction to the label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`. If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

The unindexed form is permitted for the ARM instruction set only. In it, <option> is a coprocessor option, written as an integer 0-255 enclosed in { } and encoded in `imm8`.

The pre-UAL syntax `LDC<c>L` is equivalent to `LDCL<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        NullCheckIfThumbEE(15);
        offset_addr = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
        address = if index then offset_addr else Align(PC,4);
        repeat
            Coproc_SendLoadedWord(MemA[address,4], cp, ThisInstr());  address = address + 4;
        until Coproc_DoneLoading(cp, ThisInstr());
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDC{2}{L}<c><q> <coproc>, <CRd>, <label>`	P = 1、W = 0 の通常の形式
`LDC{2}{L}<c><q> <coproc>, <CRd>, [PC,#+/-<imm>]`	P = 1、W = 0 の代替形式
`LDC{2}{L}<c><q> <coproc>, <CRd>, [PC], <option>`	P = 0、U = 1、W = 0 のインデクスなしの形式

各項目の説明については以下を参照して下さい。

`2`	指定した場合、エンコード T2/A2 が選択されます。省略した場合、エンコード T1/A1 が選択されます。
`L`	指定した場合、エンコードの D == 1 の形式が選択されます。省略した場合、D == 0 の形式が選択されます。
`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `LDC2` は常に無条件命令です。
`<coproc>`	コプロセッサの名前。標準の汎用コプロセッサ名は、p0、p1、…、p15 です。
`<CRd>`	コプロセッサのデスティネーションレジスタ
`<label>`	`<Rt>` にロードされるリテラルデータ項目のラベル。アセンブラは、この命令の `Align(PC,4)` の値からこのラベルまでのオフセットに必要な値を計算します。オフセットに許容される値は、－1020 ～ 1020 までの 4 の倍数です。 オフセットが 0 または正の場合、`imm32` はオフセットと等しく、`add == TRUE` になります。 オフセットが負の場合、`imm32` は負のオフセットと等しく、`add == FALSE` になります。

代替構文では、通常の構文では指定できない 0 の減算など、オフセットおよびイミディエートオフセットの加算または減算を個別に指定することが許可されます。詳細については、*UAL 命令構文でのラベルの使用* P. A4-5 を参照して下さい。

インデクスなしの形式は、ARM 命令セットでのみ許可されます。この形式では、〈option〉はコプロセッサオプションであり、{} で囲まれた 0 ～ 255 の整数として書き込まれ、`imm8` にエンコードされます。

UAL 以前の構文の `LDC<c>L` は、`LDCL<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        NullCheckIfThumbEE(15);
        offset_addr = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
        address = if index then offset_addr else Align(PC,4);
        repeat
            Coproc_SendLoadedWord(MemA[address,4], cp, ThisInstr());  address = address + 4;
        until Coproc_DoneLoading(cp, ThisInstr());
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.53 LDM / LDMIA / LDMFD

Load Multiple (Increment After) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the highest of those locations can optionally be written back to the base register. The registers loaded can include the PC, causing a branch to a loaded address. Related system instructions are *LDM (user registers)* on page B6-7 and *LDM (exception return)* on page B6-5.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7 (not in ThumbEE)

```
LDM<c> <Rn>!,<registers>        <Rn> not included in <registers>
LDM<c> <Rn>,<registers>         <Rn> included in <registers>
```

15 14 13 12	11	10 9 8 7 6 5 4 3 2 1 0
1 1 0 0	1	Rn	register_list

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = (registers<n> == '0');
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Encoding T2 ARMv6T2, ARMv7

```
LDM<c>.W <Rn>{!},<registers>
```

15 14 13 12	11	10	9	8 7 6	5	4	3 2 1 0	15	14	13	12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0	1	0	0	0 1 0	W	1	Rn	P	M	(0)	register_list

```
if W == '1' && Rn == '1101' then SEE POP;
n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDM<c> <Rn>{!},<registers>
```

31 30 29 28	27 26 25 24 23 22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0 0 1 0	W	1	Rn	register_list

```
if W == '1' && Rn == '1101' && BitCount(register_list) >= 2 then SEE POP;
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```

Assembler syntax

```
LDM<c><q>  <Rn>{!}, <registers>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.53 LDM/LDMIA/LDMFD

LDM、LDMIA、LDMFD（複数ロード（ポストインクリメント））は、ベースレジスタからのアドレスを使用して、連続するメモリ位置から複数のレジスタをロードします。連続するメモリ位置は、このアドレスから開始されます。オプションとして、ロードされる最上位メモリ位置の次のアドレスをベースレジスタに書き戻すことができます。ロードするレジスタには、PC を含めることができます。これにより、ロードされるアドレスへの分岐が発生します。関連するシステム命令は *LDM（ユーザレジスタ）* P. B6-7 と *LDM（例外からの復帰）* P. B6-5 です。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7（ThumbEE 以外）

```
LDM<c> <Rn>!,<registers>        <Rn> は <registers> に含まれない
LDM<c> <Rn>,<registers>         <Rn> は <registers> に含まれる
```

15 14 13 12	11	10 9 8 7 6 5 4 3 2 1 0
1 1 0 0	1	Rn	register_list

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = (registers<n> == '0');
if BitCount(registers) < 1 then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

```
LDM<c>.W <Rn>{!},<registers>
```

15 14 13 12	11	10	9	8 7 6	5	4	3 2 1 0	15	14	13	12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0	1	0	0	0 1 0	W	1	Rn	P	M	(0)	register_list

```
if W == '1' && Rn == '1101' then SEE POP;
n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDM<c> <Rn>{!},<registers>
```

31 30 29 28	27 26 25 24 23 22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0 0 1 0	W	1	Rn	register_list

```
if W == '1' && Rn == '1101' && BitCount(register_list) >= 2 then SEE POP;
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```

アセンブラ構文

```
LDM<c><q>  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<c><q> 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> ベースレジスタ。このレジスタには SP を指定できます。SP と！を指定した場合は、*POP* P. A8-246 に説明されているように命令が扱われます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<Rn>` The base register. SP can be used. If it is the SP and ! is specified, the instruction is treated as described in *POP* on page A8-246.

`!` Causes the instruction to write a modified value back to `<Rn>`. Encoded as W = 1. If ! is omitted, the instruction does not change `<Rn>` in this way. Encoded as W = 0.

`<registers>` Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address.

Encoding T2 does not support a list containing only one register. If an `LDMIA` instruction with just one register `<Rt>` in the list is assembled to Thumb and encoding T1 is not available, it is assembled to the equivalent `LDR<c><q> <Rt>,[<Rn>]{,#4}` instruction.

The SP can be in the list in ARM code, but not in Thumb code. However, ARM instructions that include the SP in the list are deprecated.

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. In ARMv5T and above, this is an interworking branch, see *Pseudocode details of operations on ARM core registers* on page A2-12. In Thumb code, if the PC is in the list:

- the LR must not be in the list
- the instruction must be either outside any IT block, or the last instruction in an IT block.

ARM instructions that include both the LR and the PC in the list are deprecated.

Instructions with the base register in the list and ! specified are only available in the ARM instruction set before ARMv7, and the use of such instructions is deprecated. The value of the base register after such an instruction is UNKNOWN.

`LDMIA` and `LDMFD` are pseudo-instructions for `LDM`. `LDMFD` refers to its use for popping data from Full Descending stacks.

The pre-UAL syntaxes `LDM<c>IA` and `LDM<c>FD` are equivalent to `LDM<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`!` 命令の実行によって変更された値を `<Rn>` に書き戻すように指示します。W = 1 としてエンコードされます。! を省略した場合は、命令によって `<Rn>` は変更されません。W = 0 としてエンコードされます。

`<registers>` ロードされる 1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストです。最下位のメモリアドレスに置かれているデータは最も小さい番号のレジスタにロードされ、以下順に、最上位のメモリアドレスのデータは最も大きい番号のレジスタにロードされます。

エンコード T2 では、1 つのレジスタしか含まないリストはサポートされません。`LDMIA` 命令のレジスタリストに `<Rt>` しか含まれていない場合、この命令が Thumb にアセンブルされ、エンコード T1 が使用できない場合は、等価な `LDR<c><q> <Rt>,[<Rn>]{,#4}` 命令にアセンブルされます。

ARM コードではリストに SP を含めることができますが、Thumb コードではできません。しかし、リストに SP を含める ARM 命令は推奨されません。

リストに PC を含めることができます。リストに PC が含まれている場合、PC にロードされるアドレスに命令が分岐します。ARMv5T およびそれ以降では、これはインターワーキング分岐です。詳細については、*ARM コアレジスタ演算の擬似コードの詳細* P. A2- 12 を参照して下さい。Thumb コードでは、リストに PC が含まれている場合に次の条件が適用されます。

- LR をリストに含めることはできません。
- この命令は、IT ブロック外部に配置するか、IT ブロック内部の最後の命令にする必要があります。

リストに LR と PC の両方を含める ARM 命令は推奨されません。

ベースレジスタがリストに含まれ、! が指定されている命令は、ARMv7 より前の ARM 命令セットでのみ使用できます。このような命令の使用は推奨されません。このような命令の後のベースレジスタの値は未知です。

`LDMIA` および `LDMFD` は、`LDM` の擬似命令です。`LDMFD` は、フル下降スタックからデータをポップするために使用することを意味しています。

UAL 以前の構文の `LDM<c>IA` および `LDM<c>FD` は、`LDM<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.54 LDMDA / LDMFA

Load Multiple Decrement After (Load Multiple Full Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register. The registers loaded can include the PC, causing a branch to a loaded address.

Related system instructions are *LDM (user registers)* on page B6-7 and *LDM (exception return)* on page B6-5.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDMDA<c> <Rn>{!},<registers>
```

31 30 29 28	27 26 25 24 23 22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0 0 0 0	W	1	Rn	register_list

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.54 LDMDA/LDMFA

LDMDA、LDMFA（複数ロード（ポストデクリメント））（複数ロード、フル上昇）は、ベースレジスタからのアドレスを使用して、連続するメモリ位置から複数のレジスタをロードします。連続するメモリ位置は、このアドレスで終了します。オプションとして、ロードされる最下位メモリ位置の 1 つ前のアドレスをベースレジスタに書き戻すことができます。ロードするレジスタには、PC を含めることができます。これにより、ロードされるアドレスへの分岐が発生します。

関連するシステム命令は *LDM（ユーザレジスタ）* P. B6-7 と *LDM（例外からの復帰）* P. B6-5 です。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDMDA<c> <Rn>{!},<registers>
```

31 30 29 28	27 26 25 24 23 22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0 0 0 0	W	1	Rn	register_list

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
LDMDA<c><q>  <Rn>{!}, <registers>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The base register. SP can be used.

! Causes the instruction to write a modified value back to <Rn>. Encoded as W = 1.

If ! is omitted, the instruction does not change <Rn> in this way. Encoded as W = 0.

<registers> Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address.

The SP can be in the list. However, instructions that include the SP in the list are deprecated.

The PC can be in the list. If it is, the instruction branches to the address (data) loaded to the PC. In ARMv5T and above, this branch is an interworking branch, see *Pseudocode details of operations on ARM core registers* on page A2-12.

Instructions that include both the LR and the PC in the list are deprecated.

Instructions with the base register in the list and ! specified are only available before ARMv7, and the use of such instructions is deprecated. The value of the base register after such an instruction is UNKNOWN.

LDMFA is a pseudo-instruction for LDMDA, referring to its use for popping data from Full Ascending stacks.

The pre-UAL syntaxes LDM<c>DA and LDM<c>FA are equivalent to LDMDA<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
LDMDA<c>&lt;q&gt;  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<c><q> 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> ベースレジスタ。このレジスタには SP を指定できます。

! 命令の実行によって変更された値を <Rn> に書き戻すように指示します。W = 1 としてエンコードされます。

! を省略した場合は、命令によって <Rn> は変更されません。W = 0 としてエンコードされます。

<registers> ロードされる 1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストです。最下位のメモリアドレスに置かれているデータは最も小さい番号のレジスタにロードされ、以下順に、最上位のメモリアドレスのデータは最も大きい番号のレジスタにロードされます。

リストに SP を含めることができます。しかし、リストに SP を含める命令は推奨されません。

リストに PC を含めることができます。リストに PC が含まれている場合、PC にロードされるアドレス（データ）に命令が分岐します。ARMv5T およびそれ以降では、これはインターワーキング分岐です。詳細については、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 を参照して下さい。

リストに LR と PC の両方を含める命令は推奨されません。

ベースレジスタがリストに含まれ、! が指定されている命令は、ARMv7 より前でのみ使用できます。このような命令の使用は推奨されません。このような命令の後のベースレジスタの値は未知です。

LDMFA は LDMDA の擬似命令で、フル上昇スタックからデータをポップするために使用されることを示します。

UAL 以前の構文の LDM<c>DA および LDM<c>FA は、LDMDA<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.55 LDMDB / LDMEA

Load Multiple Decrement Before (Load Multiple Empty Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the lowest of those locations can optionally be written back to the base register. The registers loaded can include the PC, causing a branch to a loaded address.

Related system instructions are *LDM (user registers)* on page B6-7 and *LDM (exception return)* on page B6-5.

Encoding T1 ARMv6T2, ARMv7

```
LDMDB<c> <Rn>{!},<registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	0	0	W	1	Rn				P	M	(0)	register_list												

```
n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDMDB<c> <Rn>{!},<registers>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	0	1	0	0	W	1	Rn				register_list															

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```

Assembler syntax

```
LDMDB<c><q>  <Rn>{!}, <registers>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rn>` The base register. The SP can be used.

`!` Causes the instruction to write a modified value back to `<Rn>`. Encoded as W = 1.

If `!` is omitted, the instruction does not change `<Rn>` in this way. Encoded as W = 0.

`<registers>` Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address.

Encoding T1 does not support a list containing only one register. If an `LDMDB` instruction with just one register `<Rt>` in the list is assembled to Thumb, it is assembled to the equivalent `LDR<c><q> <Rt>,[<Rn>,#-4]{!}` instruction.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.55 LDMDB/LDMEA

LDMDB、LDMEA（複数ロード（プリデクリメント））（複数ロード、空上昇）は、ベースレジスタからのアドレスを使用して、連続するメモリ位置から複数のレジスタをロードします。連続するメモリ位置は、このアドレスの 1 つ前で終了します。オプションとして、ロードされる最下位メモリ位置のアドレスをベースレジスタに書き戻すことができます。ロードするレジスタには、PC を含めることができます。これにより、ロードされるアドレスへの分岐が発生します。

関連するシステム命令は *LDM（ユーザレジスタ）* P. B6-7 と *LDM（例外からの復帰）* P. B6-5 です。

エンコード T1 ARMv6T2、ARMv7

```
LDMDB<c> <Rn>{!},<registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	0	0	W	1	Rn				P	M	(0)	register_list												

```
n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDMDB<c> <Rn>{!},<registers>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	0	1	0	0	W	1	Rn				register_list															

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```

アセンブラ構文

```
LDMDB<c><q>  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`!` 命令の実行によって変更された値を `<Rn>` に書き戻すように指示します。W = 1 としてエンコードされます。

`!` を省略した場合は、命令によって `<Rn>` は変更されません W = 0 としてエンコードされます。

`<registers>` ロードされる 1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストです。最下位のメモリアドレスに置かれているデータは最も小さい番号のレジスタにロードされ、以下順に、最上位のメモリアドレスのデータは最も大きい番号のレジスタにロードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The SP can be in the list in ARM code, but not in Thumb code. However, ARM instructions that include the SP in the list are deprecated.

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. In ARMv5T and above, this is an interworking branch, see *Pseudocode details of operations on ARM core registers* on page A2-12. In Thumb code, if the PC is in the list:

- the LR must not be in the list
- the instruction must be either outside any IT block, or the last instruction in an IT block.

ARM instructions that include both the LR and the PC in the list are deprecated.

Instructions with the base register in the list and ! specified are only available in the ARM instruction set before ARMv7, and the use of such instructions is deprecated. The value of the base register after such an instruction is UNKNOWN.

LDMEA is a pseudo-instruction for LDMDB, referring to its use for popping data from Empty Ascending stacks.

The pre-UAL syntaxes LDM<c>DB and LDM<c>EA are equivalent to LDMDB<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

エンコード T1 では、1 つのレジスタのみを含むリストはサポートされません。リストに <Rt> のみを含む LDMDB 命令が Thumb にアセンブルされた場合、等価な LDR<c><q> <Rt>,[<Rn>,#-4]{!} 命令にアセンブルされます。

ARM コードではリストに SP を含めることができますが、Thumb コードではできません。しかし、リストに SP を含める ARM 命令は推奨されません。

リストに PC を含めることができます。リストに PC が含まれている場合、PC にロードされるアドレスに命令が分岐します。ARMv5T およびそれ以降では、これはインターワーキング分岐です。詳細については、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 を参照して下さい。Thumb コードでは、リストに PC が含まれている場合に次の条件が適用されます。

- LR をリストに含めることはできません。
- この命令は、IT ブロック外部に配置するか、IT ブロック内部の最後の命令にする必要があります。

リストに LR と PC の両方を含める ARM 命令は推奨されません。

ベースレジスタがリストに含まれ、! が指定されている命令は、ARMv7 より前の ARM 命令セットでのみ使用できます。このような命令の使用は推奨されません。このような命令の後のベースレジスタの値は未知です。

LDMEA は LDMDB の擬似命令で、空上昇スタックからデータをポップするために使用されることを示します。

UAL 以前の構文の LDM<c>DB および LDM<c>EA は、LDMDB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.56 LDMIB / LDMED

Load Multiple Increment Before loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register. The registers loaded can include the PC, causing a branch to a loaded address.

Related system instructions are *LDM (user registers)* on page B6-7 and *LDM (exception return)* on page B6-5.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDMIB<c> <Rn>{!},<registers>
```

31 30 29 28	27 26 25 24 23 22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0 1 1 0	W	1	Rn	register_list

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.56 LDMIB/LDMED

LDMIB、LDMED（複数ロード（プリインクリメント））は、ベースレジスタからのアドレスを使用して、連続するメモリ位置から複数のレジスタをロードします。連続するメモリ位置は、このアドレスの 1 つ後から開始されます。オプションとして、ロードされる最後のメモリ位置のアドレスをベースレジスタに書き戻すことができます。ロードするレジスタには、PC を含めることができます。これにより、ロードされるアドレスへの分岐が発生します。

関連するシステム命令は *LDM（ユーザレジスタ）* P. B6-7 と *LDM（例外からの復帰）* P. B6-5 です。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDMIB<c> <Rn>{!},<registers>
```

31 30 29 28	27 26 25 24 23 22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0 1 1 0	W	1	Rn	register_list

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
LDMIB<c><q>  <Rn>{!}, <registers>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The base register. The SP can be used.

! Causes the instruction to write a modified value back to <Rn>. Encoded as W = 1.

If ! is omitted, the instruction does not change <Rn> in this way. Encoded as W = 0.

<registers> Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address.

The SP can be in the list. However, instructions that include the SP in the list are deprecated.

The PC can be in the list. If it is, the instruction branches to the address (data) loaded to the PC. In ARMv5T and above, this branch is an interworking branch, see *Pseudocode details of operations on ARM core registers* on page A2-12.

Instructions that include both the LR and the PC in the list are deprecated.

Instructions with the base register in the list and ! specified are only available before ARMv7, and the use of such instructions is deprecated. The value of the base register after such an instruction is UNKNOWN.

LDMED is a pseudo-instruction for LDMIB, referring to its use for popping data from Empty Descending stacks.

The pre-UAL syntaxes LDM<c>IB and LDM<c>ED are equivalent to LDMIB<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
LDMIB<c><q>  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<c><q>. 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn>. ベースレジスタ。このレジスタには SP を指定できます。

!. 命令の実行によって変更された値を <Rn> に書き戻すように指示します。W = 1 としてエンコードされます。

! を省略した場合は、命令によって <Rn> は変更されません。W = 0 としてエンコードされます。

<registers>. ロードされる 1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストです。最下位のメモリアドレスに置かれているデータは最も小さい番号のレジスタにロードされ、以下順に、最上位のメモリアドレスのデータは最も大きい番号のレジスタにロードされます。

リストに SP を含めることができます。しかし、リストに SP を含める命令は推奨されません。

リストに PC を含めることができます。リストに PC が含まれている場合、PC にロードされるアドレス（データ）に命令が分岐します。ARMv5T およびそれ以降では、これはインターワーキング分岐です。詳細については、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 を参照して下さい。

リストに LR と PC の両方を含める命令は推奨されません。

ベースレジスタがリストに含まれ、! が指定されている命令は、ARMv7 より前でのみ使用できます。このような命令の使用は推奨されません。このような命令の後のベースレジスタの値は未知です。

LDMED は LDMIB の擬似命令で、空下降スタックからデータをポップするために使用されることを示します。

UAL 以前の構文の LDM<c>IB および LDM<c>ED は、LDMIB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.57 LDR (immediate, Thumb)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>, [<Rn>{,#<imm>}]
```

15	14	13	12	11	10 9 8 7 6	5 4 3	2 1 0
0	1	1	0	1	imm5	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>,[SP{,#<imm>}]
```

15	14	13	12	11	10 9 8	7 6 5 4 3 2 1 0
1	0	0	1	1	Rt	imm8

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T3 ARMv6T2, ARMv7

```
LDR<c>.W <Rt>,[<Rn>{,#<imm12>}]
```

15	14	13	12	11	10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
1	1	1	1	1	0 0	0	1	1 0	1	Rn	Rt	imm12

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T4 ARMv6T2, ARMv7

```
LDR<c> <Rt>,[<Rn>,#-<imm8>]
LDR<c> <Rt>,[<Rn>],#+/-<imm8>
LDR<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
1	1	1	1	1	0 0	0	0	1 0	1	Rn	Rt	1	P U W	imm8

```
if Rn == '1111' then SEE LDR (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRT;
if Rn == '1101' && P == '0' && U == '1' && W == '1' && imm8 == '00000100' then SEE POP;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.57 LDR（イミディエート、Thumb）

LDR（レジスタロード（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ワードをロードし、レジスタに書き込みます。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス*P. A8- 13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LDR<c> <Rt>, [<Rn>{,#<imm>}]
```

15	14	13	12	11	10 9 8 7 6	5 4 3	2 1 0
0	1	1	0	1	imm5	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LDR<c> <Rt>,[SP{,#<imm>}]
```

15	14	13	12	11	10 9 8	7 6 5 4 3 2 1 0
1	0	0	1	1	Rt	imm8

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T3 ARMv6T2、ARMv7

```
LDR<c>.W <Rt>,[<Rn>{,#<imm12>}]
```

15	14	13	12	11	10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
1	1	1	1	1	0 0	0	1	1 0	1	Rn	Rt	imm12

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード T4 ARMv6T2、ARMv7

```
LDR<c> <Rt>,[<Rn>,#-<imm8>]
LDR<c> <Rt>,[<Rn>],#+/-<imm8>
LDR<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
1	1	1	1	1	0 0	0	0	1 0	1	Rn	Rt	1	P U W	imm8

```
if Rn == '1111' then SEE LDR (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRT;
if Rn == '1101' && P == '0' && U == '1' && W == '1' && imm8 == '00000100' then SEE POP;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDR<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDR<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDR<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. In ARMv5T and above, this branch is an interworking branch, see *Pseudocode details of operations on ARM core registers* on page A2-12.

`<Rn>` The base register. The SP can be used. For PC use see *LDR (literal)* on page A8-122.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0. Values are:

Encoding T1 multiples of 4 in the range 0-124

Encoding T2 multiples of 4 in the range 0-1020

Encoding T3 any value in the range 0-4095

Encoding T4 any value in the range 0-255.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    elsif UnalignedSupport() || address<1:0> = '00' then
        R[t] = data;
    else R[t] = bits(32) UNKNOWN; // Can only apply before ARMv7
```

Exceptions

Data Abort.

ThumbEE instruction

ThumbEE has additional `LDR` (immediate) encodings. For details see *LDR (immediate)* on page A9-19.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDR<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDR<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDR<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ。このレジスタには SP を指定できます。命令が IT ブロック外部に配置されている場合、または IT ブロックの最後の命令の場合は、PC を指定できます。PC を指定した場合、PC にロードされるアドレス（データ）に命令が分岐します。ARMv5T およびそれ以降では、これはインターワーキング分岐です。詳細については、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 を参照して下さい。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC の指定については、*LDR（リテラル）* P. A8-122 を参照して下さい。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。値は次のとおりです。

エンコード T1 4 の倍数で 0 ～ 124

エンコード T2 4 の倍数で 0 ～ 1020

エンコード T3 0 ～ 4095 の任意の値

エンコード T4 0 ～ 255 の任意の値

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    elsif UnalignedSupport() || address<1:0> = '00' then
        R[t] = data;
    else R[t] = bits(32) UNKNOWN; // Can only apply before ARMv7
```

例外

データアボート

ThumbEE 命令

ThumbEE には、追加の `LDR`（イミディエート）エンコードがあります。詳細については、*LDR（イミディエート）* P. A9-19 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.58 LDR (immediate, ARM)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>,[<Rn>{,#+/-<imm12>}]
LDR<c> <Rt>,[<Rn>],#+/-<imm12>
LDR<c> <Rt>,[<Rn>,#+/-<imm12>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	P	U	0	W	1	Rn	Rt	imm12

```
if Rn == '1111' then SEE LDR (literal);
if P == '0' && W == '1' then SEE LDRT;
if Rn == '1101' && P == '0' && U == '1' && W == '0' && imm12 == '000000000100' then SEE POP;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && n == t then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.58 LDR（イミディエート、ARM）

LDR（レジスタロード（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ワードをロードし、レジスタに書き込みます。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス*P. A8- 13 を参照して下さい。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDR<c> <Rt>,[<Rn>{,#+/-<imm12>}]
LDR<c> <Rt>,[<Rn>],#+/-<imm12>
LDR<c> <Rt>,[<Rn>,#+/-<imm12>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	P	U	0	W	1	Rn	Rt	imm12

```
if Rn == '1111' then SEE LDR (literal);
if P == '0' && W == '1' then SEE LDRT;
if Rn == '1101' && P == '0' && U == '1' && W == '0' && imm12 == '000000000100' then SEE POP;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && n == t then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDR<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDR<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDR<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register. The SP or the PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. In ARMv5T and above, this branch is an interworking branch, see *Pseudocode details of operations on ARM core registers* on page A2-12.

`<Rn>` The base register. The SP can be used. For PC use see *LDR (literal)* on page A8-122.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0. Any value in the range 0-4095 is permitted.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    elsif UnalignedSupport() || address<1:0> = '00' then
        R[t] = data;
    else // Can only apply before ARMv7
        R[t] = ROR(data, 8*UInt(address<1:0>));
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDR<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDR<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDR<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ。SP または PC を指定できます。PC を指定した場合、PC にロードされるアドレス（データ）に命令が分岐します。ARMv5T およびそれ以降では、これはインターワーキング分岐です。詳細については、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 を参照して下さい。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC の指定については、*LDR（リテラル）* P. A8-122 を参照して下さい。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。0 ～ 4095 の任意の値が許可されます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    elsif UnalignedSupport() || address<1:0> = '00' then
        R[t] = data;
    else // Can only apply before ARMv7
        R[t] = ROR(data, 8*UInt(address<1:0>));
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.59 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>,<label>
```

15 14 13 12 11	10 9 8	7 6 5 4 3 2 1 0
0 1 0 0 1	Rt	imm8

```
t = UInt(Rt);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

Encoding T2 ARMv6T2, ARMv7

```
LDR<c>.W <Rt>,<label>
LDR<c>.W <Rt>,[PC,#-0]
```

Special case

15 14 13 12 11	10 9	8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1	0 0	0	U	1 0 1	1 1 1 1	Rt	imm12

```
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>,<label>
LDR<c> <Rt>,[PC,#-0]
```

Special case

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	(1)	U	0	(0)	1	1 1 1 1	Rt	imm12

```
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

Assembler syntax

```
LDR<c><q>  <Rt>, <label>                Normal form
LDR<c><q>  <Rt>, [PC, #+/-<imm>]        Alternative form
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. In ARMv5T and above, this branch is an interworking branch, see *Pseudocode details of operations on ARM core registers* on page A2-12.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.59 LDR（リテラル）

LDR（レジスタロード（リテラル））は、PC の値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ワードをロードし、レジスタに書き込みます。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LDR<c> <Rt>,<label>
```

15 14 13 12 11	10 9 8	7 6 5 4 3 2 1 0
0 1 0 0 1	Rt	imm8

```
t = UInt(Rt);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

エンコード T2 ARMv6T2、ARMv7

```
LDR<c>.W <Rt>,<label>
LDR<c>.W <Rt>,[PC,#-0]
```

特殊なケース

15 14 13 12 11	10 9	8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1	0 0	0	U	1 0 1	1 1 1 1	Rt	imm12

```
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDR<c> <Rt>,<label>
LDR<c> <Rt>,[PC,#-0]
```

特殊なケース

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	(1)	U	0	(0)	1	1 1 1 1	Rt	imm12

```
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

アセンブラ構文

```
LDR<c><q>  <Rt>, <label>                通常の形式
LDR<c><q>  <Rt>, [PC, #+/-<imm>]        代替形式
```

各項目の説明については以下を参照して下さい。

`<c><q>` 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ。このレジスタには SP を指定できます。命令が IT ブロック外部に配置されている場合、または IT ブロックの最後の命令の場合は、PC を指定できます。PC を指定した場合、PC にロードされるアドレス（データ）に命令が分岐します。ARMv5T およびそれ以降では、これはインターワーキング分岐です。詳細については、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC,4)` value of this instruction to the label. Permitted values of the offset are:

Encoding T1 multiples of four in the range -1020 to 1020

Encoding T2 or A1 any value in the range -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`. Negative offset is not available in encoding T1.

Note

In code examples in this manual, the syntax `=<value>` is used for the label of a memory word whose contents are constant and equal to `<value>`. The actual syntax for such a label is assembler-dependent.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(15);
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,4];
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    elsif UnalignedSupport() || address<1:0> = '00' then
        R[t] = data;
    else // Can only apply before ARMv7
        if CurrentInstrSet() == InstrSet_ARM then
            R[t] = ROR(data, 8*UInt(address<1:0>));
        else
            R[t] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<label>` `<Rt>` にロードされるリテラルデータ項目のラベル。アセンブラは、この命令の `Align(PC,4)` の値からこのラベルまでのオフセットに必要な値を計算します。オフセットに許容される値の範囲は次のとおりです。

エンコード T1 4 の倍数で－1020 ～ 1020

エンコード T2 または A1 －4095 ～ 4095 の任意の値

オフセットが 0 または正の場合、`imm32` はオフセットと等しく、`add == TRUE` になります。

オフセットが負の場合、`imm32` は負のオフセットと等しく、`add == FALSE` となります。エンコード T1 では、負のオフセットを使用できません。

注

本書のコード例では、構文 `= <value>` は、内容が定数である、`<value>` に等しいメモリワードのラベルに使用されます。このようなラベル用の実際の構文は、アセンブラに依存します。

代替構文では、通常の構文では指定できない 0 の減算など、オフセットおよびイミディエートオフセットの加算または減算を個別に指定することが許可されます。詳細については、*UAL 命令構文でのラベルの使用* P. A4-5 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(15);
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,4];
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    elsif UnalignedSupport() || address<1:0> = '00' then
        R[t] = data;
    else // Can only apply before ARMv7
        if CurrentInstrSet() == InstrSet_ARM then
            R[t] = ROR(data, 8*UInt(address<1:0>));
        else
            R[t] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.60 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses, see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>,[<Rn>,<Rm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	Rm			Rn			Rt		

```
if CurrentInstrSet() == InstrSet_ThumbEE then SEE "Modified operation in ThumbEE";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
LDR<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	1	Rn				Rt				0	0	0	0	0	0	imm2		Rm			

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(m) then UNPREDICTABLE;
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>,[<Rn>,+/-<Rm>{, <shift>}]{!}
LDR<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	P	U	0	W	1	Rn				Rt				imm5					type		0	Rm			

```
if P == '0' && W == '1' then SEE LDRT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

Modified operation in ThumbEE See *LDR (register)* on page A9-9

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.60 LDR（レジスタ）

LDR（レジスタロード（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 ワードをロードし、レジスタに書き込みます。オフセットレジスタの値は、オプションでシフトすることができます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LDR<c> <Rt>,[<Rn>,<Rm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	Rm			Rn			Rt		

```
if CurrentInstrSet() == InstrSet_ThumbEE then SEE "Modified operation in ThumbEE";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
LDR<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	1	Rn				Rt				0	0	0	0	0	0	imm2		Rm			

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(m) then UNPREDICTABLE;
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDR<c> <Rt>,[<Rn>,+/-<Rm>{, <shift>}]{!}
LDR<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	P	U	0	W	1	Rn				Rt				imm5					type		0	Rm			

```
if P == '0' && W == '1' then SEE LDRT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

ThumbEE での変更された動作 *LDR（レジスタ）* P. A9- 9 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDR<c><q> <Rt>, [<Rn>, +/-<Rm>{, <shift>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDR<c><q> <Rt>, [<Rn>, +/-<Rm>{, <shift>}]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDR<c><q> <Rt>, [<Rn>], +/-<Rm>{, <shift>}`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. In ARMv5T and above, this branch is an interworking branch, see *Pseudocode details of operations on ARM core registers* on page A2-12.

`<Rn>` The base register. The SP can be used. The PC can be used only in the ARM instruction set.

`+/-` Is + or omitted if the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, add == FALSE).

`<Rm>` The offset that is optionally shifted and applied to the value of `<Rn>` to form the address.

`<shift>` The shift to apply to the value read from `<Rm>`. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted. For encoding T2, `<shift>` can only be omitted, encoded as imm2 = `0b00`, or `LSL #<imm>` with `<imm>` = 1, 2, or 3, and `<imm>` encoded in imm2. For encoding A1, see *Shifts applied to a register* on page A8-10.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    elsif UnalignedSupport() || address<1:0> = '00' then
        R[t] = data;
    else // Can only apply before ARMv7
        if CurrentInstrSet() == InstrSet_ARM then
            R[t] = ROR(data, 8*UInt(address<1:0>));
        else
            R[t] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDR<c><q> <Rt>, [<Rn>, +/-<Rm>{, <shift>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDR<c><q> <Rt>, [<Rn>, +/-<Rm>{, <shift>}]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDR<c><q> <Rt>, [<Rn>], +/-<Rm>{, <shift>}`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` 詳細については、*標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>` デスティネーションレジスタ。このレジスタには SP を指定できます。命令が IT ブロック外部に配置されている場合、または IT ブロックの最後の命令の場合は、PC を指定できます。PC を指定した場合、PC にロードされるアドレス（データ）に命令が分岐します。ARMv5T およびそれ以降では、これはインターワーキング分岐です。詳細については、*ARM コアレジスタ演算の擬似コードの詳細* P. A2- 12 を参照して下さい。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC は、ARM 命令セットでのみ指定できます。

`+/-` オプションでシフトされる `<Rm>` の値をベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、add == FALSE）は、「-」にします。

`<Rm>` アドレス生成のためにオプションでシフトされ、`<Rn>` の値に適用されるオフセット

`<shift>` `<Rm>` から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。エンコード T2 では、`<shift>` のみ省略することができ、imm2 = `0b00`、または `LSL #<imm>`（`<imm>` = 1、2、または 3）としてエンコードされ、`<imm>` は imm2 にエンコードされます。エンコード A1 については、*レジスタに適用されるシフト* P. A8- 10 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    elsif UnalignedSupport() || address<1:0> = '00' then
        R[t] = data;
    else // Can only apply before ARMv7
        if CurrentInstrSet() == InstrSet_ARM then
            R[t] = ROR(data, 8*UInt(address<1:0>));
        else
            R[t] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.61 LDRB (immediate, Thumb)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LDRB<c> <Rt>,[<Rn>{,#<imm5>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 ARMv6T2, ARMv7

```
LDRB<c>.W <Rt>,[<Rn>{,#<imm12>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	0	1	Rn				Rt				imm12											

```
if Rt == '1111' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

```
LDRB<c> <Rt>,[<Rn>,#-<imm8>]
LDRB<c> <Rt>,[<Rn>],#+/-<imm8>
LDRB<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	1	Rn				Rt				1	P	U	W	imm8							

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRBT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.61 LDRB（イミディエート、Thumb）

LDRB（レジスタロード バイト（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LDRB<c> <Rt>,[<Rn>{,#<imm5>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T2 ARMv6T2、ARMv7

```
LDRB<c>.W <Rt>,[<Rn>{,#<imm12>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	0	1	Rn				Rt				imm12											

```
if Rt == '1111' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

```
LDRB<c> <Rt>,[<Rn>,#-<imm8>]
LDRB<c> <Rt>,[<Rn>],#+/-<imm8>
LDRB<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	1	Rn				Rt				1	P	U	W	imm8							

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRBT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRB<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRB<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used. For PC use see *LDRB (literal)* on page A8-130.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0. Values are:

Encoding T1	any value in the range 0-31
Encoding T2	any value in the range 0-4095
Encoding T3	any value in the range 0-255.

The pre-UAL syntax `LDR<c>B` is equivalent to `LDRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDRB<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDRB<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC の指定については、*LDRB（リテラル）* P. A8-130 を参照して下さい。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。値は次のとおりです。

エンコード T1	0 ～ 31 の任意の値
エンコード T2	0 ～ 4095 の任意の値
エンコード T3	0 ～ 255 の任意の値

UAL 以前の構文の `LDR<c>B` は、`LDRB<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.62 LDRB (immediate, ARM)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRB<c> <Rt>,[<Rn>{,#+/-<imm12>}]
LDRB<c> <Rt>,[<Rn>],#+/-<imm12>
LDRB<c> <Rt>,[<Rn>,#+/-<imm12>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	P	U	1	W	1	Rn	Rt	imm12

```
if Rn == '1111' then SEE LDRB (literal);
if P == '0' && W == '1' then SEE LDRBT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.62 LDRB（イミディエート、ARM）

LDRB（レジスタロード バイト（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRB<c> <Rt>,[<Rn>{,#+/-<imm12>}]
LDRB<c> <Rt>,[<Rn>],#+/-<imm12>
LDRB<c> <Rt>,[<Rn>,#+/-<imm12>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	P	U	1	W	1	Rn	Rt	imm12

```
if Rn == '1111' then SEE LDRB (literal);
if P == '0' && W == '1' then SEE LDRBT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRB<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRB<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used. For PC use see *LDRB (literal)* on page A8-130.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0. Any value in the range 0-4095 is permitted.

The pre-UAL syntax `LDR<c>B` is equivalent to `LDRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDRB<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDRB<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC の指定については、*LDRB（リテラル）* P. A8-130 を参照して下さい。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。0 ～ 4095 の任意の値が許可されます。

UAL 以前の構文の `LDR<c>B` は、`LDRB<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.63 LDRB (literal)

Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
LDRB<c> <Rt>,<label>
LDRB<c> <Rt>,[PC,#-0]                                   Special case
```

15 14 13 12	11 10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1	0 0	0	U	0 0	1	1 1 1 1	Rt	imm12

```
if Rt == '1111' then SEE PLD;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRB<c> <Rt>,<label>
LDRB<c> <Rt>,[PC,#-0]                                   Special case
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	(1)	U	1	(0)	1	1 1 1 1	Rt	imm12

```
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.63 LDRB（リテラル）

LDRB（レジスタロード バイト（リテラル））は、PC の値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。メモリアクセスの詳細については、*メモリアクセス*P. A8-13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
LDRB<c> <Rt>,<label>
LDRB<c> <Rt>,[PC,#-0]                                   特殊なケース
```

15 14 13 12	11 10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1	0 0	0	U	0 0	1	1 1 1 1	Rt	imm12

```
if Rt == '1111' then SEE PLD;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRB<c> <Rt>,<label>
LDRB<c> <Rt>,[PC,#-0]                                   特殊なケース
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	(1)	U	1	(0)	1	1 1 1 1	Rt	imm12

```
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRB<c><q> <Rt>, <label>`	Normal form
`LDRB<c><q> <Rt>, [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC,4)` value of this instruction to the label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

The pre-UAL syntax `LDR<c>B` is equivalent to `LDRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(15);
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = ZeroExtend(MemU[address,1], 32);
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRB<c><q> <Rt>, <label>`	通常の形式
`LDRB<c><q> <Rt>, [PC, #+/-<imm>]`	代替形式

各項目の説明については以下を参照して下さい。

`<c><q>` 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<label>` `<Rt>` にロードされるリテラルデータ項目のラベル。アセンブラは、この命令の `Align(PC,4)` の値からこのラベルまでのオフセットに必要な値を計算します。オフセットに許容される値の範囲は－4095 ～ 4095 です。

オフセットが 0 または正の場合、`imm32` はオフセットと等しく、`add == TRUE` になります。

オフセットが負の場合、`imm32` は負のオフセットと等しく、`add == FALSE` になります。

代替構文では、通常の構文では指定できない 0 の減算など、オフセットおよびイミディエートオフセットの加算または減算を個別に指定することが許可されます。詳細については、*UAL 命令構文でのラベルの使用* P. A4-5 を参照して下さい。

UAL 以前の構文の `LDR<c>B` は、`LDRB<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(15);
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = ZeroExtend(MemU[address,1], 32);
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.64 LDRB (register)

Load Register Byte (register) calculates an address from abase register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LDRB<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	1 1 0	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
LDRB<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11	10 9 8 7 6	5 4 3 2 1 0
1 1 1 1	1 0 0	0	0	0	0	1	Rn	Rt	0	0 0 0 0 0	imm2 Rm

```
if Rt == '1111' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRB<c> <Rt>,[<Rn>,+/-<Rm>{, <shift>}]{!}
LDRB<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 1 1	P	U	1	W	1	Rn	Rt	imm5	type	0	Rm

```
if P == '0' && W == '1' then SEE LDRBT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.64 LDRB（レジスタ）

LDRB（レジスタロード バイト（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。オフセットレジスタの値は、オプションでシフトすることができます。メモリアクセスの詳細については、*メモリアクセス*P. A8- 13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LDRB<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	1 1 0	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
LDRB<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11	10 9 8 7 6	5 4 3 2 1 0
1 1 1 1	1 0 0	0	0	0	0	1	Rn	Rt	0	0 0 0 0 0	imm2 Rm

```
if Rt == '1111' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRB<c> <Rt>,[<Rn>,+/-<Rm>{, <shift>}]{!}
LDRB<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 1 1	P	U	1	W	1	Rn	Rt	imm5	type	0	Rm

```
if P == '0' && W == '1' then SEE LDRBT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRB<c><q> <Rt>, [<Rn>, +/-<Rm>{, <shift>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRB<c><q> <Rt>, [<Rn>, +/-<Rm>{, <shift>}]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRB<c><q> <Rt>, [<Rn>], +/-<Rm>{, <shift>}`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used. The PC can be used only in the ARM instruction set.

`+/-` Is + or omitted if the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, add == FALSE).

`<Rm>` Contains the offset that is optionally shifted and applied to the value of `<Rn>` to form the address.

`<shift>` The shift to apply to the value read from `<Rm>`. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted. For encoding T2, `<shift>` can only be omitted, encoded as imm2 = `0b00`, or `LSL #<imm>` with `<imm>` = 1, 2, or 3, and `<imm>` encoded in imm2. For encoding A1, see *Shifts applied to a register* on page A8-10.

The pre-UAL syntax `LDR<c>B` is equivalent to `LDRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1],32);
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRB<c><q> <Rt>, [<Rn>, +/-<Rm>{, <shift>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDRB<c><q> <Rt>, [<Rn>, +/-<Rm>{, <shift>}]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDRB<c><q> <Rt>, [<Rn>], +/-<Rm>{, <shift>}`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC は、ARM 命令セットでのみ指定できます。

`+/-` オプションでシフトされる `<Rm>` の値をベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、add == FALSE）は、「-」にします。

`<Rm>` アドレス生成のためにオプションでシフトされて、`<Rn>` の値に適用されるオフセットを格納します。

`<shift>` `<Rm>` から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。エンコード T2 では、`<shift>` として省略（imm2 = `0b00` としてエンコードされます）、または `LSL #<imm>` で `<imm>` = 1、2、3（`<imm>` が imm2 にエンコードされます）のみが使用可能です。エンコード A1 については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

UAL 以前の構文の `LDR<c>B` は、`LDRB<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1],32);
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.65 LDRBT

Load Register Byte Unprivileged loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page A8-13.

The memory access is restricted as if the processor were running in User mode. (This makes no difference if the processor is actually running in User mode.)

The Thumb instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The ARM instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

Encoding T1 ARMv6T2, ARMv7

```
LDRBT<c> <Rt>,[<Rn>,#<imm8>]
```

15 14 13 12 11	10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 1	0 0	0	0	0 0	1	Rn	Rt	1	1 1 0	imm8

```
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRBT<c> <Rt>,[<Rn>],#+/-<imm12>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	0	U	1	1	1	Rn	Rt	imm12

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

Encoding A2 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRBT<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 1 1	0	U	1	1	1	Rn	Rt	imm5	type	0	Rm

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && m == n then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.65 LDRBT

LDRBT（非特権レジスタロード バイト）は、メモリから 1 バイトをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。メモリアクセスの詳細については、*メモリアクセス*P. A8-13 を参照して下さい。

プロセッサがユーザモードで実行されている場合と同様にメモリアクセスが制限されます（実際にプロセッサがユーザモードで実行されている場合は、何も変わりません）。

Thumb 命令では、オフセット アドレシングモードを使用します。このモードでは、メモリアクセスに使用されるアドレスをベースレジスタの値とイミディエートオフセットから計算し、ベースレジスタの値は変更しません。

ARM 命令では、ポストインデクス アドレシングモードを使用します。このモードでは、ベースレジスタの値をメモリアクセス用のアドレスとして使用し、ベースレジスタの値とオフセットから新しいアドレスを計算して、ベースレジスタに書き戻します。オフセットは、イミディエート値またはオプションとしてシフトされるレジスタの値です。

エンコード T1 ARMv6T2、ARMv7

```
LDRBT<c> <Rt>,[<Rn>,#<imm8>]
```

15 14 13 12 11	10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 1	0 0	0	0	0 0	1	Rn	Rt	1	1 1 0	imm8

```
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRBT<c> <Rt>,[<Rn>],#+/-<imm12>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	0	U	1	1	1	Rn	Rt	imm12

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

エンコード A2 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRBT<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 1 1	0	U	1	1	1	Rn	Rt	imm5	type	0	Rm

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && m == n then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRBT<c><q> <Rt>, [<Rn> {, #<imm>}]`	Offset: Thumb only
`LDRBT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	Post-indexed: ARM only
`LDRBT<c><q> <Rt>, [<Rn>], +/-<Rm> {, <shift>}`	Post-indexed: ARM only

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used.

`+/-` Is + or omitted if `<imm>` or the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, `add == FALSE`).

`<imm>` The immediate offset applied to the value of `<Rn>`. Values are 0-255 for encoding T1, and 0-4095 for encoding A1. `<imm>` can be omitted, meaning an offset of 0.

`<Rm>` Contains the offset that is optionally shifted and applied to the value of `<Rn>` to form the address.

`<shift>` The shift to apply to the value read from `<Rm>`. If omitted, no shift is applied. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

The pre-UAL syntax `LDR<c>BT` is equivalent to `LDRBT<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then Shift(R[m], shift_t, shift_n, APSR.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = ZeroExtend(MemU_unpriv[address,1],32);
    if postindex then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRBT<c><q> <Rt>, [<Rn> {, #<imm>}]`	オフセット：Thumb のみ
`LDRBT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	ポストインデクス：ARM のみ
`LDRBT<c><q> <Rt>, [<Rn>], +/-<Rm> {, <shift>}`	ポストインデクス：ARM のみ

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`+/-` `<imm>` またはオプションとしてシフトされる `<Rm>` の値をベースレジスタの値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、`add == FALSE`）は、「-」にします。

`<imm>` `<Rn>` の値に適用されるイミディエートオフセット。値の範囲は、エンコード T1 では 0 〜 255、エンコード A1 では 0 〜 4095 です。`<imm>` は省略可能で、その場合オフセットは 0 と見なされます。

`<Rm>` アドレス生成のためにオプションとしてシフトされ、`<Rn>` の値に適用されるオフセットを格納します。

`<shift>` `<Rm>` から読み出された値に適用されるシフト。省略した場合、シフトは適用されません。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8- 10 を参照して下さい。

UAL 以前の構文の `LDR<c>BT` は、`LDRBT<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then Shift(R[m], shift_t, shift_n, APSR.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = ZeroExtend(MemU_unpriv[address,1],32);
    if postindex then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.66 LDRD (immediate)

Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
LDRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm>}]
LDRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm>
LDRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm>]!
```

15 14 13 12 11	10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0 1	0 0	P	U	1	W	1	Rn	Rt	Rt2	imm8

```
if P == '0' && W == '0' then SEE "Related encodings";
if Rn == '1111' then SEE LDRD (literal);
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if BadReg(t) || BadReg(t2) || t == t2 then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
LDRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm8>}]
LDRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm8>
LDRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm8>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	1	W	0	Rn	Rt	imm4H	1 1 0 1	imm4L

```
if Rn == '1111' then SEE LDRD (literal);
if Rt<0> == '1' then UNDEFINED;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;
```

Related encodings See *Load/store dual, load/store exclusive, table branch* on page A6-24

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.66 LDRD（イミディエート）

LDRD（レジスタロード デュアル（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 2 ワードをロードし、2 つのレジスタに書き込みます。オフセットアドレシング、ポスト インデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
LDRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm>}]
LDRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm>
LDRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm>]!
```

15 14 13 12 11	10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0 1	0 0	P	U	1	W	1	Rn	Rt	Rt2	imm8

```
if P == '0' && W == '0' then SEE "Related encodings";
if Rn == '1111' then SEE LDRD (literal);
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if BadReg(t) || BadReg(t2) || t == t2 then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
LDRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm8>}]
LDRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm8>
LDRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm8>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	1	W	0	Rn	Rt	imm4H	1 1 0 1	imm4L

```
if Rn == '1111' then SEE LDRD (literal);
if Rt<0> == '1' then UNDEFINED;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;
```

関連エンコード *デュアルロード／ストア、排他ロード／ストア、テーブル分岐* P. A6-24 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRD<c><q> <Rt>, <Rt2>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRD<c><q> <Rt>, <Rt2>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRD<c><q> <Rt>, <Rt2>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The first destination register. For an ARM instruction, `<Rt>` must be even-numbered and not R14.

`<Rt2>` The second destination register. For an ARM instruction, `<Rt2>` must be `<R(t+1)>`.

`<Rn>` The base register. The SP can be used. For PC use see *LDRD (literal)* on page A8-138.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0. Values are:

Encoding T1 multiples of 4 in the range 0-1020

Encoding A1 any value in the range 0-255.

The pre-UAL syntax `LDR<c>D` is equivalent to `LDRD<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = MemA[address,4];
    R[t2] = MemA[address+4,4];
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRD<c><q> <Rt>, <Rt2>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDRD<c><q> <Rt>, <Rt2>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDRD<c><q> <Rt>, <Rt2>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` 最初のデスティネーションレジスタ。ARM 命令の場合、`<Rt>` には R14 以外の偶数番号のレジスタのみが使用できます。

`<Rt2>` 2 番目のデスティネーションレジスタ。ARM 命令の場合、`<Rt2>` は `<R(t+1)>` でなければなりません。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC の指定については、*LDRD（リテラル）* P. A8-138 を参照して下さい。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。値は次のとおりです。

エンコード T1 4 の倍数で 0 ～ 1020

エンコード A1 0 ～ 255 の任意の値

UAL 以前の構文の `LDR<c>D` は、`LDRD<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = MemA[address,4];
    R[t2] = MemA[address+4,4];
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.67 LDRD (literal)

Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
LDRD<c> <Rt>,<Rt2>,<label>
LDRD<c> <Rt>,<Rt2>,[PC,#-0]                    Special case
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	P	U	1	(0)	1	1	1	1	1	Rt				Rt2				imm8							

```
if P == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);
imm32 = ZeroExtend(imm8:'00', 32);  add = (U == '1');
if BadReg(t) || BadReg(t2) || t == t2 then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
LDRD<c> <Rt>,<Rt2>,<label>
LDRD<c> <Rt>,<Rt2>,[PC,#-0]                    Special case
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	(1)	U	1	(0)	0	1	1	1	1	Rt				imm4H				1	1	0	1	imm4L			

```
if Rt<0> == '1' then UNDEFINED;
t = UInt(Rt);  t2 = t+1;  imm32 = ZeroExtend(imm4H:imm4L, 32);  add = (U == '1');
if t2 == 15 then UNPREDICTABLE;
```

Related encodings See *Load/store dual, load/store exclusive, table branch* on page A6-24

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.67 LDRD（リテラル）

LDRD（レジスタロード デュアル（リテラル））は、PC の値とイミディエートオフセットからアドレスを計算して、そのメモリから 2 ワードをロードし、2 つのレジスタに書き込みます。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
LDRD<c> <Rt>,<Rt2>,<label>
LDRD<c> <Rt>,<Rt2>,[PC,#-0]                    特殊なケース
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	P	U	1	(0)	1	1	1	1	1	Rt				Rt2				imm8							

```
if P == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);
imm32 = ZeroExtend(imm8:'00', 32);  add = (U == '1');
if BadReg(t) || BadReg(t2) || t == t2 then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
LDRD<c> <Rt>,<Rt2>,<label>
LDRD<c> <Rt>,<Rt2>,[PC,#-0]                    特殊なケース
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	(1)	U	1	(0)	0	1	1	1	1	Rt				imm4H				1	1	0	1	imm4L			

```
if Rt<0> == '1' then UNDEFINED;
t = UInt(Rt);  t2 = t+1;  imm32 = ZeroExtend(imm4H:imm4L, 32);  add = (U == '1');
if t2 == 15 then UNPREDICTABLE;
```

関連エンコード *デュアルロード／ストア、排他ロード／ストア、テーブル分岐* P. A6-24 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRD<c><q> <Rt>, <Rt2>, <label>`	Normal form
`LDRD<c><q> <Rt>, <Rt2>, [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The first destination register. For an ARM instruction, `<Rt>` must be even-numbered and not R14.

`<Rt2>` The second destination register. For an ARM instruction, `<Rt2>` must be `<R(t+1)>`.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC,4)` value of this instruction to the label. Permitted values of the offset are:

Encoding T1 multiples of 4 in the range -1020 to 1020

Encoding A1 any value in the range -255 to 255.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

The pre-UAL syntax `LDR<c>D` is equivalent to `LDRD<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(15);
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    R[t] = MemA[address,4];
    R[t2] = MemA[address+4,4];
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRD<c><q> <Rt>, <Rt2>, <label>`	通常の形式
`LDRD<c><q> <Rt>, <Rt2>, [PC, #+/-<imm>]`	代替形式

各項目の説明については以下を参照して下さい。

`<c><q>` 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` 最初のデスティネーションレジスタ。ARM 命令の場合、`<Rt>` には R14 以外の偶数番号のレジスタのみが使用できます。

`<Rt2>` 2 番目のデスティネーションレジスタ。ARM 命令の場合、`<Rt2>` は `<R(t+1)>` でなければなりません。

`<label>` `<Rt>` にロードされるリテラルデータ項目のラベル。アセンブラは、この命令の `Align(PC,4)` の値からこのラベルまでのオフセットに必要な値を計算します。オフセットに許容される値の範囲は次のとおりです。

エンコード T1 4 の倍数で − 1020 ～ 1020

エンコード A1 − 255 ～ 255 の任意の値

オフセットが 0 または正の場合、`imm32` はオフセットと等しく、`add == TRUE` になります。

オフセットが負の場合、`imm32` は負のオフセットと等しく、`add == FALSE` になります。

代替構文では、通常の構文では指定できない 0 の減算など、オフセットおよびイミディエートオフセットの加算または減算を個別に指定することが許可されます。詳細については、*UAL 命令構文でのラベルの使用* P. A4-5 を参照して下さい。

UAL 以前の構文の `LDR<c>D` は、`LDRD<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(15);
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    R[t] = MemA[address,4];
    R[t2] = MemA[address+4,4];
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.68 LDRD (register)

Load Register Dual (register) calculates an address from abase register value and a register offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
LDRD<c> <Rt>,<Rt2>,[<Rn>,+/-<Rm>]{!}
LDRD<c> <Rt>,<Rt2>,[<Rn>],+/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	0	W	0	Rn	Rt	(0) (0) (0) (0)	1 1 0 1	Rm

```
if Rt<0> == '1' then UNDEFINED;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if t2 == 15 || m == 15 || m == t || m == t2 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.68 LDRD（レジスタ）

LDRD（レジスタロード デュアル（レジスタ））は、ベースレジスタの値とレジスタオフセットからアドレスを計算して、そのメモリから 2 ワードをロードし、2 つのレジスタに書き込みます。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
LDRD<c> <Rt>,<Rt2>,[<Rn>,+/-<Rm>]{!}
LDRD<c> <Rt>,<Rt2>,[<Rn>],+/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	0	W	0	Rn	Rt	(0) (0) (0) (0)	1 1 0 1	Rm

```
if Rt<0> == '1' then UNDEFINED;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if t2 == 15 || m == 15 || m == t || m == t2 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRD<c><q> <Rt>, <Rt2>, [<Rn>, +/-<Rm>]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRD<c><q> <Rt>, <Rt2>, [<Rn>, +/-<Rm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRD<c><q> <Rt>, <Rt2>, [<Rn>], +/-<Rm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The first destination register. This register must be even-numbered and not R14.

`<Rt2>` The second destination register. This register must be `<R(t+1)>`.

`<Rn>` The base register. The SP or the PC can be used.

`+/-` Is + or omitted if the value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`).

`<Rm>` Contains the offset that is applied to the value of `<Rn>` to form the address.

The pre-UAL syntax `LDR<c>D` is equivalent to `LDRD<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    R[t] = MemA[address,4];
    R[t2] = MemA[address+4,4];
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRD<c><q> <Rt>, <Rt2>, [<Rn>, +/-<Rm>]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDRD<c><q> <Rt>, <Rt2>, [<Rn>, +/-<Rm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDRD<c><q> <Rt>, <Rt2>, [<Rn>], +/-<Rm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` 最初のデスティネーションレジスタ。R14 以外の偶数番号のレジスタのみが使用できます。

`<Rt2>` 2 番目のデスティネーションレジスタ。このレジスタは、`<R(t+1)>` でなければなりません。

`<Rn>` ベースレジスタ。SP または PC を指定できます。

`+/-` `<Rm>` の値をベースレジスタの値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合 (`add == FALSE`) は、「−」にします。

`<Rm>` アドレス生成のために `<Rn>` の値に適用されるオフセットを格納します。

UAL 以前の構文の `LDR<c>D` は、`LDRD<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    R[t] = MemA[address,4];
    R[t2] = MemA[address+4,4];
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.69 LDREX

Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register and:

- if the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing processor in a shared monitor
- causes the executing processor to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores* on page A3-12. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
LDREX<c> <Rt>,[<Rn>{,#<imm>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	1	0	1	Rn				Rt				(1)	(1)	(1)	(1)	imm8							

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if BadReg(t) || n == 15 then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
LDREX<c> <Rt>,[<Rn>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	0	1	Rn				Rt				(1)	(1)	(1)	(1)	1	0	0	1	(1)	(1)	(1)	(1)

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if t == 15 || n == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.69 LDREX

LDREX（排他レジスタロード）は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ワードをロードしてレジスタに書き込み、次の動作を実行します。

- アドレスに共用メモリ属性が設定されている場合は、共有モニタで実行中のプロセッサのために、その物理アドレスを排他的アクセスとしてマークします。
- 実行中のプロセッサが、ローカルモニタでアクティブな排他アクセスを示すようにします。

共有メモリのサポートの詳細については、*同期化とセマフォ* P. A3-12 を参照して下さい。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
LDREX<c> <Rt>,[<Rn>{,#<imm>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	1	0	1	Rn				Rt				(1)	(1)	(1)	(1)	imm8							

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if BadReg(t) || n == 15 then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
LDREX<c> <Rt>,[<Rn>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	0	1	Rn				Rt				(1)	(1)	(1)	(1)	1	0	0	1	(1)	(1)	(1)	(1)

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if t == 15 || n == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
LDREX<c><q>  <Rt>, [<Rn> {,#<imm>}]
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rt> The destination register.

<Rn> The base register. The SP can be used.

<imm> The immediate offset added to the value of <Rn> to form the address. <imm> can be omitted, meaning an offset of 0. Values are:

Encoding T1 multiples of 4 in the range 0-1020

Encoding A1 omitted or 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + imm32;
    SetExclusiveMonitors(address,4);
    R[t] = MemA[address,4];
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
LDREX<c><q>  <Rt>, [<Rn> {,#<imm>}]
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rt> デスティネーションレジスタ

<Rn> ベースレジスタ。このレジスタには SP を指定できます。

<imm> アドレス生成のために <Rn> の値に加算されるイミディエートオフセット。<imm> は省略可能で、その場合オフセットは 0 と見なされます。値は次のとおりです。

エンコード T1 4 の倍数で 0 ～ 1020

エンコード A1 省略されるか、0

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + imm32;
    SetExclusiveMonitors(address,4);
    R[t] = MemA[address,4];
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.70 LDREXB

Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- if the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing processor in a shared monitor
- causes the executing processor to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores* on page A3-12. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv7

```
LDREXB<c> <Rt>, [<Rn>]
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 1	0 0 0 1 1 0	1	Rn	Rt	(1) (1) (1) (1)	0 1 0 0	(1) (1) (1) (1)

```
t = UInt(Rt);  n = UInt(Rn);
if BadReg(t) || n == 15 then UNPREDICTABLE;
```

Encoding A1 ARMv6K, ARMv7

```
LDREXB<c> <Rt>, [<Rn>]
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	1 1 1 0	1	Rn	Rt	(1) (1) (1) (1)	1 0 0 1	(1) (1) (1) (1)

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.70 LDREXB

LDREXB（排他レジスタロード バイト）は、ベースレジスタの値からアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードにゼロ拡張してレジスタに書き込み、次の動作を実行します。

- アドレスに共用メモリ属性が設定されている場合は、共有モニタで実行中のプロセッサのために、その物理アドレスを排他的アクセスとしてマークします。
- 実行中のプロセッサが、ローカルモニタでアクティブな排他アクセスを示すようにします。

共有メモリのサポートの詳細については、*同期化とセマフォ* P. A3-12 を参照して下さい。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv7

```
LDREXB<c> <Rt>, [<Rn>]
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 1	0 0 0 1 1 0	1	Rn	Rt	(1) (1) (1) (1)	0 1 0 0	(1) (1) (1) (1)

```
t = UInt(Rt);  n = UInt(Rn);
if BadReg(t) || n == 15 then UNPREDICTABLE;
```

エンコード A1 ARMv6K、ARMv7

```
LDREXB<c> <Rt>, [<Rn>]
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	1 1 1 0	1	Rn	Rt	(1) (1) (1) (1)	1 0 0 1	(1) (1) (1) (1)

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
LDREXB<c><q>  <Rt>, [<Rn>]
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rt> The destination register.

<Rn> The base register. The SP can be used.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    SetExclusiveMonitors(address,1);
    R[t] = ZeroExtend(MemA[address,1], 32);
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
LDREXB<c><q>  <Rt>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

<c><q> 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rt> デスティネーションレジスタ

<Rn> ベースレジスタ。このレジスタには SP を指定できます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    SetExclusiveMonitors(address,1);
    R[t] = ZeroExtend(MemA[address,1], 32);
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.71 LDREXD

Load Register Exclusive Doubleword derives an address from a base register value, loads a 64-bit doubleword from memory, writes it to two registers and:

- if the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing processor in a shared monitor
- causes the executing processor to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores* on page A3-12. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv7

```
LDREXD<c> <Rt>,<Rt2>,[<Rn>]
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 1	0 0 0 1 1 0	1	Rn	Rt	Rt2	0 1 1 1	(1) (1) (1) (1)

```
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if BadReg(t) || BadReg(t2) || t == t2 || n == 15 then UNPREDICTABLE;
```

Encoding A1 ARMv6K, ARMv7

```
LDREXD<c> <Rt>,<Rt2>,[<Rn>]
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	1 1 0 1	1	Rn	Rt	(1) (1) (1) (1)	1 0 0 1	(1) (1) (1) (1)

```
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);
if Rt<0> = '1' || Rt == '1110' || n == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.71 LDREXD

LDREXD（排他レジスタロード ダブルワード）は、ベースレジスタの値からアドレスを計算して、そのメモリから 64 ビットダブルワードをロードし、2 つのレジスタに書き込み、次の動作を実行します。

- アドレスに共用メモリ属性が設定されている場合は、共有モニタで実行中のプロセッサのために、その物理アドレスを排他的アクセスとしてマークします。
- 実行中のプロセッサが、ローカルモニタでアクティブな排他アクセスを示すようにします。

共有メモリのサポートの詳細については、*同期化とセマフォ* P. A3-12 を参照して下さい。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv7

```
LDREXD<c> <Rt>,<Rt2>,[<Rn>]
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 1	0 0 0 1 1 0	1	Rn	Rt	Rt2	0 1 1 1	(1) (1) (1) (1)

```
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if BadReg(t) || BadReg(t2) || t == t2 || n == 15 then UNPREDICTABLE;
```

エンコード A1 ARMv6K、ARMv7

```
LDREXD<c> <Rt>,<Rt2>,[<Rn>]
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	1 1 0 1	1	Rn	Rt	(1) (1) (1) (1)	1 0 0 1	(1) (1) (1) (1)

```
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);
if Rt<0> = '1' || Rt == '1110' || n == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
LDREXD<c><q>  <Rt>, <Rt2>, [<Rn>]
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rt> The first destination register. For an ARM instruction, <Rt> must be even-numbered and not R14.

<Rt2> The second destination register. For an ARM instruction, <Rt2> must be <R(t+1)>.

<Rn> The base register. The SP can be used.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    SetExclusiveMonitors(address,8);
    value = MemA[address,8];
    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R[t]  = if BigEndian() then value<63:32> else value<31:0>;
    R[t2] = if BigEndian() then value<31:0> else value<63:32>;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
LDREXD<c><q>  <Rt>, <Rt2>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

<c><q> 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rt> 最初のデスティネーションレジスタ。ARM 命令の場合、<Rt> には R14 以外の偶数番号のレジスタのみが使用できます。

<Rt2> 2 番目のデスティネーションレジスタ。ARM 命令の場合、<Rt2> は <R(t+1)> でなければなりません。

<Rn> ベースレジスタ。このレジスタには SP を指定できます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    SetExclusiveMonitors(address,8);
    value = MemA[address,8];
    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R[t]  = if BigEndian() then value<63:32> else value<31:0>;
    R[t2] = if BigEndian() then value<31:0> else value<63:32>;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.72 LDREXH

Load Register Exclusive Halfword derives an address from abase register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- if the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing processor in a shared monitor
- causes the executing processor to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores* on page A3-12. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv7

```
LDREXH<c> <Rt>, [<Rn>]
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 1	0 0 0 1 1 0	1	Rn	Rt	(1) (1) (1) (1)	0 1 0 1	(1) (1) (1) (1)

```
t = UInt(Rt);  n = UInt(Rn);
if BadReg(t) || n == 15 then UNPREDICTABLE;
```

Encoding A1 ARMv6K, ARMv7

```
LDREXH<c> <Rt>, [<Rn>]
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	1 1 1 1	1	Rn	Rt	(1) (1) (1) (1)	1 0 0 1	(1) (1) (1) (1)

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.72 LDREXH

LDREXH（排他レジスタロード ハーフワード）は、ベースレジスタの値からアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードにゼロ拡張してレジスタに書き込み、次の動作を実行します。

- アドレスに共用メモリ属性が設定されている場合は、共有モニタで実行中のプロセッサのために、その物理アドレスを排他的アクセスとしてマークします。
- 実行中のプロセッサが、ローカルモニタでアクティブな排他アクセスを示すようにします。

共有メモリのサポートの詳細については、*同期化とセマフォ* P. A3-12 を参照して下さい。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv7

```
LDREXH<c> <Rt>, [<Rn>]
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 1	0 0 0 1 1 0	1	Rn	Rt	(1) (1) (1) (1)	0 1 0 1	(1) (1) (1) (1)

```
t = UInt(Rt);  n = UInt(Rn);
if BadReg(t) || n == 15 then UNPREDICTABLE;
```

エンコード A1 ARMv6K、ARMv7

```
LDREXH<c> <Rt>, [<Rn>]
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	1 1 1 1	1	Rn	Rt	(1) (1) (1) (1)	1 0 0 1	(1) (1) (1) (1)

```
t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
LDREXH<c><q>  <Rt>, [<Rn>]
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rt>`	The destination register.
`<Rn>`	The base register. The SP can be used.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    SetExclusiveMonitors(address,2);
    R[t] = ZeroExtend(MemA[address,2], 32);
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
LDREXH<c><q>  <Rt>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

`<c><q>`	詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rt>`	デスティネーションレジスタ
`<Rn>`	ベースレジスタ。このレジスタには SP を指定できます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    SetExclusiveMonitors(address,2);
    R[t] = ZeroExtend(MemA[address,2], 32);
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.73 LDRH (immediate, Thumb)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LDRH<c> <Rt>,[<Rn>{,#<imm>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 ARMv6T2, ARMv7

```
LDRH<c>.W <Rt>,[<Rn>{,#<imm12>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	1	1	Rn				Rt				imm12											

```
if Rt == '1111' then SEE "Unallocated memory hints";
if Rn == '1111' then SEE LDRH (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

```
LDRH<c> <Rt>,[<Rn>,#-<imm8>]
LDRH<c> <Rt>,[<Rn>],#+/-<imm8>
LDRH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	1	Rn				Rt				1	P	U	W	imm8							

```
if Rn == '1111' then SEE LDRH (literal);
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Unallocated memory hints";
if P == '1' && U == '1' && W == '0' then SEE LDRHT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```

Unallocated memory hints See *Load halfword, memory hints* on page A6-26

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.73 LDRH（イミディエート、Thumb）

LDRH（レジスタロード ハーフワード（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LDRH<c> <Rt>,[<Rn>{,#<imm>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T2 ARMv6T2、ARMv7

```
LDRH<c>.W <Rt>,[<Rn>{,#<imm12>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	1	1	Rn				Rt				imm12											

```
if Rt == '1111' then SEE "Unallocated memory hints";
if Rn == '1111' then SEE LDRH (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

```
LDRH<c> <Rt>,[<Rn>,#-<imm8>]
LDRH<c> <Rt>,[<Rn>],#+/-<imm8>
LDRH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	1	Rn				Rt				1	P	U	W	imm8							

```
if Rn == '1111' then SEE LDRH (literal);
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Unallocated memory hints";
if P == '1' && U == '1' && W == '0' then SEE LDRHT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```

未割り当てメモリヒント *ハーフワードロード、メモリヒント* P. A6-26 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRH<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRH<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used. For PC use see *LDRH (literal)* on page A8-154.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` The immediate offset used to form the address. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0. Values are:

Encoding T1	multiples of 2 in the range 0-62
Encoding T2	any value in the range 0-4095
Encoding T3	any value in the range 0-255.

The pre-UAL syntax `LDR<c>H` is equivalent to `LDRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = ZeroExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDRH<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDRH<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC の指定については、*LDRH（リテラル）* P. A8-154 を参照して下さい。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。値は次のとおりです。

エンコード T1	2 の倍数で 0 ～ 62
エンコード T2	0 ～ 4095 の任意の値
エンコード T3	0 ～ 255 の任意の値

UAL 以前の構文の `LDR<c>H` は、`LDRH<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = ZeroExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.74 LDRH (immediate, ARM)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRH<c> <Rt>,[<Rn>{,#+/-<imm8>}]
LDRH<c> <Rt>,[<Rn>],#+/-<imm8>
LDRH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	1	W	1	Rn	Rt	imm4H	1 0 1 1	imm4L

```
if Rn == '1111' then SEE LDRH (literal);
if P == '0' && W == '1' then SEE LDRHT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.74 LDRH（イミディエート、ARM）

LDRH（レジスタロード ハーフワード（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRH<c> <Rt>,[<Rn>{,#+/-<imm8>}]
LDRH<c> <Rt>,[<Rn>],#+/-<imm8>
LDRH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	1	W	1	Rn	Rt	imm4H	1 0 1 1	imm4L

```
if Rn == '1111' then SEE LDRH (literal);
if P == '0' && W == '1' then SEE LDRHT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRH<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRH<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used. For PC use see *LDRH (literal)* on page A8-154.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` The immediate offset used to form the address. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0. Any value in the range 0-255 is permitted.

The pre-UAL syntax `LDR<c>H` is equivalent to `LDRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = ZeroExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDRH<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDRH<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC の指定については、*LDRH（リテラル）* P. A8-154 を参照して下さい。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。0 ～ 255 の任意の値が許可されます。

UAL 以前の構文の `LDR<c>H` は、`LDRH<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = ZeroExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.75 LDRH (literal)

Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
LDRH<c> <Rt>,<label>
LDRH<c> <Rt>,[PC,#-0]                                   Special case
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	U	0	1	1	1	1	1	1	Rt				imm12											

```
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRH<c> <Rt>,<label>
LDRH<c> <Rt>,[PC,#-0]                                   Special case
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	(1)	U	1	(0)	1	1	1	1	1	Rt				imm4H				1	0	1	1	imm4L			

```
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);  add = (U == '1');
if t == 15 then UNPREDICTABLE;
```

Unallocated memory hints See *Load halfword, memory hints* on page A6-26

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.75 LDRH（リテラル）

LDRH（レジスタロード ハーフワード（リテラル））は、PC の値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
LDRH<c> <Rt>,<label>
LDRH<c> <Rt>,[PC,#-0]                                   特殊なケース
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	U	0	1	1	1	1	1	1	Rt				imm12											

```
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRH<c> <Rt>,<label>
LDRH<c> <Rt>,[PC,#-0]                                   特殊なケース
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	(1)	U	1	(0)	1	1	1	1	1	Rt				imm4H				1	0	1	1	imm4L			

```
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);  add = (U == '1');
if t == 15 then UNPREDICTABLE;
```

未割り当てメモリヒント *ハーフワードロード、メモリヒント* P. A6- 26 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRH<c><q> <Rt>, <label>`	Normal form
`LDRH<c><q> <Rt>, [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC,4)` value of the `ADR` instruction to this label. Permitted values of the offset are:

Encoding T1 any value in the range -4095 to 4095

Encoding A1 any value in the range -255 to 255.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

The pre-UAL syntax `LDR<c>H` is equivalent to `LDRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(15);
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    if UnalignedSupport() || address<0> = '0' then
        R[t] = ZeroExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRH<c><q> <Rt>, <label>`	通常の形式
`LDRH<c><q> <Rt>, [PC, #+/-<imm>]`	代替形式

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<label>` `<Rt>` にロードされるリテラルデータ項目のラベル。アセンブラは、`ADR` 命令の `Align(PC,4)` の値からこのラベルまでのオフセットに必要な値を計算します。オフセットに許容される値の範囲は次のとおりです。

エンコード T1 − 4095 〜 4095 の任意の値

エンコード A1 − 255 〜 255 の任意の値

オフセットが 0 または正の場合、`imm32` はオフセットと等しく、`add == TRUE` になります。

オフセットが負の場合、`imm32` は負のオフセットと等しく、`add == FALSE` になります。

代替構文では、通常の構文では指定できない 0 の減算など、オフセットおよびイミディエートオフセットの加算または減算を個別に指定することが許可されます。詳細については、*UAL 命令構文でのラベルの使用* P. A4-5 を参照して下さい。

UAL 以前の構文の `LDR<c>H` は、`LDRH<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(15);
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    if UnalignedSupport() || address<0> = '0' then
        R[t] = ZeroExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.76 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LDRH<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	1 0 1	Rm	Rn	Rt

```
if CurrentInstrSet() == InstrSet_ThumbEE then SEE "Modified operation in ThumbEE";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
LDRH<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12	11 10 9	8 7 6	5 4	3 2 1 0	15 14 13 12	11	10 9 8 7 6	5 4	3 2 1 0
1 1 1 1	0 0	0 0	0 1	1	Rn	Rt	0	0 0 0 0 0	imm2	Rm

```
if Rn == '1111' then SEE LDRH (literal);
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRH<c> <Rt>,[<Rn>,+/-<Rm>]{!}
LDRH<c> <Rt>,[<Rn>],+/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	0	W	1	Rn	Rt	(0) (0) (0) (0)	1 0 1 1	Rm

```
if P == '0' && W == '1' then SEE LDRHT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

Unallocated memory hints See *Load halfword, memory hints* on page A6-26

Modified operation in ThumbEE See *LDRH (register)* on page A9-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.76 LDRH（レジスタ）

LDRH（レジスタロード ハーフワード（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。オフセットレジスタの値は、左に 0、1、2、3 ビットシフトできます。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LDRH<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	1 0 1	Rm	Rn	Rt

```
if CurrentInstrSet() == InstrSet_ThumbEE then SEE "Modified operation in ThumbEE";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
LDRH<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12	11 10 9	8 7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8 7 6	5 4	3 2 1 0
1 1 1 1	0 0	0 0	0 1	1	Rn	Rt	0	0 0 0 0 0	imm2	Rm

```
if Rn == '1111' then SEE LDRH (literal);
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRH<c> <Rt>,[<Rn>,+/-<Rm>]{!}
LDRH<c> <Rt>,[<Rn>],+/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	0	W	1	Rn	Rt	(0) (0) (0) (0)	1 0 1 1	Rm

```
if P == '0' && W == '1' then SEE LDRHT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

未割り当てメモリヒント *ハーフワードロード、メモリヒント* P. A6-26 を参照して下さい。

ThumbEE で変更された動作 *LDRH（レジスタ）* P. A9-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRH<c><q> <Rt>, [<Rn>, <Rm>{, LSL #<imm>}]`	Offset: `index==TRUE, wback==FALSE`
`LDRH<c><q> <Rt>, [<Rn>, +/-<Rm>]`	Offset: `index==TRUE, wback==FALSE`
`LDRH<c><q> <Rt>, [<Rn>, +/-<Rm>]!`	Pre-indexed: `index==TRUE, wback==TRUE`
`LDRH<c><q> <Rt>, [<Rn>], +/-<Rm>`	Post-indexed: `index==FALSE, wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used. The PC can be used only in the ARM instruction set.

`+/-` Is + or omitted if the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, add == FALSE).

`<Rm>` Contains the offset that is optionally left shifted and added to the value of `<Rn>` to form the address.

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. Only encoding T2 is permitted, and `<imm>` is encoded in imm2.

If absent, no shift is specified and all encodings are permitted. In encoding T2, imm2 is encoded as `0b00`.

The pre-UAL syntax `LDR<c>H` is equivalent to `LDRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = ZeroExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRH<c><q> <Rt>, [<Rn>, <Rm>{, LSL #<imm>}]`	オフセット：`index==TRUE、wback==FALSE`
`LDRH<c><q> <Rt>, [<Rn>, +/-<Rm>]`	オフセット：`index==TRUE、wback==FALSE`
`LDRH<c><q> <Rt>, [<Rn>, +/-<Rm>]!`	プリインデクス：`index==TRUE、wback==TRUE`
`LDRH<c><q> <Rt>, [<Rn>], +/-<Rm>`	ポストインデクス：`index==FALSE、wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC は、ARM 命令セットでのみ指定できます。

`+/-` オプションでシフトされる `<Rm>` の値をベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、add == FALSE）は、「−」にします。

`<Rm>` アドレス生成のためにオプションで左にシフトされて、`<Rn>` の値に加算されるオフセット

`<imm>` 指定した場合、左シフト量 (1 〜 3) が `<Rm>` から読み出された値に適用されます。エンコード T2 のみ許可され、`<imm>` は imm2 にエンコードされます。

省略した場合、シフトは指定されず、すべてのエンコードが許可されます。エンコード T2 では、imm2 は `0b00` としてエンコードされます。

UAL 以前の構文の `LDR<c>H` は、`LDRH<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = ZeroExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.77 LDRHT

Load Register Halfword Unprivileged loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page A8-13.

The memory access is restricted as if the processor were running in User mode. (This makes no difference if the processor is actually running in User mode.)

The Thumb instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The ARM instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

Encoding T1 ARMv6T2, ARMv7

```
LDRHT<c> <Rt>,[<Rn>,#<imm8>]
```

15 14 13 12 11	10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 1	0 0	0	0	0 1	1	Rn	Rt	1	1 1 0	imm8

```
if Rn == '1111' then SEE LDRH (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

Encoding A1 ARMv6T2, ARMv7

```
LDRHT<c> <Rt>, [<Rn>] {, #+/-<imm8>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	0	U	1	1	1	Rn	Rt	imm4H	1 0 1 1	imm4L

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

Encoding A2 ARMv6T2, ARMv7

```
LDRHT<c> <Rt>, [<Rn>], +/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	0	U	0	1	1	Rn	Rt	(0) (0) (0) (0)	1 0 1 1	Rm

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.77 LDRHT

LDRHT（非特権レジスタロード ハーフワード）は、メモリから 1 ハーフワードをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。メモリアクセスの詳細については、*メモリアクセス*P. A8- 13 を参照して下さい。

プロセッサがユーザモードで実行されている場合と同様にメモリアクセスが制限されます（実際にプロセッサがユーザモードで実行されている場合は、何も変わりません）。

Thumb 命令では、オフセット アドレシングモードを使用します。このモードでは、メモリアクセスに使用されるアドレスを、ベースレジスタの値とイミディエートオフセットから計算し、ベースレジスタの値は変更しません。

ARM 命令では、ポストインデクス アドレシングモードを使用します。このモードでは、ベースレジスタの値をメモリアクセス用のアドレスとして使用し、ベースレジスタの値とオフセットから新しいアドレスを計算して、ベースレジスタに書き戻します。オフセットは、イミディエート値またはレジスタの値です。

エンコード T1 ARMv6T2、ARMv7

```
LDRHT<c> <Rt>,[<Rn>,#<imm8>]
```

15 14 13 12 11	10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 1	0 0	0	0	0 1	1	Rn	Rt	1	1 1 0	imm8

```
if Rn == '1111' then SEE LDRH (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

エンコード A1 ARMv6T2、ARMv7

```
LDRHT<c> <Rt>, [<Rn>] {, #+/-<imm8>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	0	U	1	1	1	Rn	Rt	imm4H	1 0 1 1	imm4L

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

エンコード A2 ARMv6T2、ARMv7

```
LDRHT<c> <Rt>, [<Rn>], +/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	0	U	0	1	1	Rn	Rt	(0) (0) (0) (0)	1 0 1 1	Rm

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRHT<c><q> <Rt>, [<Rn> {, #<imm>}]`	Offset: Thumb only
`LDRHT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	Post-indexed: ARM only
`LDRHT<c><q> <Rt>, [<Rn>], +/-<Rm>`	Post-indexed: ARM only

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used.

`+/-` Is + or omitted if `<imm>` or the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, `add == FALSE`).

`<imm>` The immediate offset applied to the value of `<Rn>`. Any value in the range 0-255 is permitted. `<imm>` can be omitted, meaning an offset of 0.

`<Rm>` Contains the offset that is applied to the value of `<Rn>` to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = ZeroExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRHT<c><q> <Rt>, [<Rn> {, #<imm>}]`	オフセット：Thumb のみ
`LDRHT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	ポストインデクス：ARM のみ
`LDRHT<c><q> <Rt>, [<Rn>], +/-<Rm>`	ポストインデクス：ARM のみ

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`+/-` `<imm>` またはオプションとしてシフトされる `<Rm>` の値をベースレジスタの値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、`add == FALSE`）は、「-」にします。

`<imm>` `<Rn>` の値に適用されるイミディエートオフセット。0 ～ 255 の任意の値が許可されます。`<imm>` は省略可能で、その場合オフセットは 0 と見なされます。

`<Rm>` アドレス生成のために `<Rn>` の値に適用されるオフセットを格納します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = ZeroExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.78 LDRSB (immediate)

Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
LDRSB<c> <Rt>,[<Rn>,#<imm12>]
```

15 14 13 12	11 10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1	1 0 0	1	1	0 0	1	Rn	Rt	imm12

```
if Rt == '1111' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

Encoding T2 ARMv6T2, ARMv7

```
LDRSB<c> <Rt>,[<Rn>,#-<imm8>]
LDRSB<c> <Rt>,[<Rn>],#+/-<imm8>
LDRSB<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15 14 13 12	11 10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
1 1 1 1	1 0 0	1	0	0 0	1	Rn	Rt	1	P U W	imm8

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRSBT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRSB<c> <Rt>,[<Rn>{,#+/-<imm8>}]
LDRSB<c> <Rt>,[<Rn>],#+/-<imm8>
LDRSB<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	1	W	1	Rn	Rt	imm4H	1 1 0 1	imm4L

```
if Rn == '1111' then SEE LDRSB (literal);
if P == '0' && W == '1' then SEE LDRSBT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.78 LDRSB（イミディエート）

LDRSB（レジスタロード符号付きバイト（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
LDRSB<c> <Rt>,[<Rn>,#<imm12>]
```

15 14 13 12	11 10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1	1 0 0	1	1	0 0	1	Rn	Rt	imm12

```
if Rt == '1111' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

```
LDRSB<c> <Rt>,[<Rn>,#-<imm8>]
LDRSB<c> <Rt>,[<Rn>],#+/-<imm8>
LDRSB<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15 14 13 12	11 10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
1 1 1 1	1 0 0	1	0	0 0	1	Rn	Rt	1	P U W	imm8

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRSBT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRSB<c> <Rt>,[<Rn>{,#+/-<imm8>}]
LDRSB<c> <Rt>,[<Rn>],#+/-<imm8>
LDRSB<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	1	W	1	Rn	Rt	imm4H	1 1 0 1	imm4L

```
if Rn == '1111' then SEE LDRSB (literal);
if P == '0' && W == '1' then SEE LDRSBT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRSB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRSB<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRSB<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used. For PC use see *LDRSB (literal)* on page A8-162.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` The immediate offset used to form the address. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0. Values are:

Encoding T1 any value in the range 0-4095

Encoding T2 or A1 any value in the range0-255.

The pre-UAL syntax `LDR<c>SB` is equivalent to `LDRSB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRSB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDRSB<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDRSB<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC の指定については、*LDRSB（リテラル）* P. A8-162 を参照して下さい。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。値は次のとおりです。

エンコード T1 0 ～ 4095 の任意の値

エンコード T2 または A1 0 ～ 255 の任意の値

UAL 以前の構文の `LDR<c>SB` は、`LDRSB<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.79 LDRSB (literal)

Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
LDRSB<c> <Rt>,<label>
LDRSB<c> <Rt>,[PC,#-0]                                  Special case
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	U	0	0	1	1	1	1	1	Rt				imm12											

```
if Rt == '1111' then SEE PLI;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRSB<c> <Rt>,<label>
LDRSB<c> <Rt>,[PC,#-0]                                  Special case
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	(1)	U	1	(0)	1	1	1	1	1	Rt				imm4H				1	1	0	1	imm4L			

```
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);  add = (U == '1');
if t == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.79 LDRSB（リテラル）

LDRSB（レジスタロード符号付きバイト（リテラル））は、PC の値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
LDRSB<c> <Rt>,<label>
LDRSB<c> <Rt>,[PC,#-0]                                  特殊なケース
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	U	0	0	1	1	1	1	1	Rt				imm12											

```
if Rt == '1111' then SEE PLI;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRSB<c> <Rt>,<label>
LDRSB<c> <Rt>,[PC,#-0]                                  特殊なケース
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	(1)	U	1	(0)	1	1	1	1	1	Rt				imm4H				1	1	0	1	imm4L			

```
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);  add = (U == '1');
if t == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRSB<c><q> <Rt>, <label>`	Normal form
`LDRSB<c><q> <Rt>, [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC,4)` value of the `ADR` instruction to this label. Permitted values of the offset are:

Encoding T1 any value in the range -4095 to 4095

Encoding A1 any value in the range -255 to 255.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

The pre-UAL syntax `LDR<c>SB` is equivalent to `LDRSB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(15);
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = SignExtend(MemU[address,1], 32);
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRSB<c><q> <Rt>, <label>`	通常の形式
`LDRSB<c><q> <Rt>, [PC, #+/-<imm>]`	代替形式

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<label>` `<Rt>` にロードされるリテラルデータ項目のラベル。アセンブラは、`ADR` 命令の `Align(PC,4)` の値からこのラベルまでのオフセットに必要な値を計算します。オフセットに許容される値の範囲は次のとおりです。

エンコード T1 − 4095 〜 4095 の任意の値

エンコード A1 − 255 〜 255 の任意の値

オフセットが 0 または正の場合、`imm32` はオフセットと等しく、`add == TRUE` になります。

オフセットが負の場合、`imm32` は負のオフセットと等しく、`add == FALSE` になります。

代替構文では、通常の構文では指定できない 0 の減算など、オフセットおよびイミディエートオフセットの加算または減算を個別に指定することが許可されます。詳細については、*UAL 命令構文でのラベルの使用* P. A4-5 を参照して下さい。

UAL 以前の構文の `LDR<c>SB` は、`LDRSB<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(15);
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = SignExtend(MemU[address,1], 32);
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.80 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LDRSB<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	0 1 1	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
LDRSB<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11	10 9 8 7 6	5 4 3 2 1 0
1 1 1 1	1 0 0	1	0	0	0	1	Rn	Rt	0	0 0 0 0 0	imm2 Rm

```
if Rt == '1111' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRSB<c> <Rt>,[<Rn>,+/-<Rm>]{!}
LDRSB<c> <Rt>,[<Rn>],+/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	0	W	1	Rn	Rt	(0) (0) (0) (0)	1 1 0 1	Rm

```
if P == '0' && W == '1' then SEE LDRSBT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.80 LDRSB（レジスタ）

LDRSB（レジスタロード符号付きバイト（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。オフセットレジスタの値は、左に 0、1、2、3 ビットシフトできます。メモリアクセスの詳細については、*メモリアクセス*P. A8- 13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LDRSB<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	0 1 1	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
LDRSB<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11	10 9 8 7 6	5 4 3 2 1 0
1 1 1 1	1 0 0	1	0	0	0	1	Rn	Rt	0	0 0 0 0 0	imm2 Rm

```
if Rt == '1111' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRSB<c> <Rt>,[<Rn>,+/-<Rm>]{!}
LDRSB<c> <Rt>,[<Rn>],+/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	0	W	1	Rn	Rt	(0) (0) (0) (0)	1 1 0 1	Rm

```
if P == '0' && W == '1' then SEE LDRSBT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

LDRSB<c><q> <Rt>, [<Rn>, <Rm>{, LSL #<imm>}]	Offset: index==TRUE, wback==FALSE
LDRSB<c><q> <Rt>, [<Rn>, +/-<Rm>]	Offset: index==TRUE, wback==FALSE
LDRSB<c><q> <Rt>, [<Rn>, +/-<Rm>]!	Pre-indexed: index==TRUE, wback==TRUE
LDRSB<c><q> <Rt>, [<Rn>], +/-<Rm>	Post-indexed: index==FALSE, wback==TRUE

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rt> The destination register.

<Rn> The base register. The SP can be used. The PC can be used only in the ARM instruction set.

+/- Is + or omitted if the optionally shifted value of <Rm> is to be added to the base register value (add == TRUE), or – if it is to be subtracted (permitted in ARM code only, add == FALSE).

<Rm> Contains the offset that is optionally left shifted and added to the value of <Rn> to form the address.

<imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. Only encoding T2 is permitted, and <imm> is encoded in imm2.

If absent, no shift is specified and all encodings are permitted. In encoding T2, imm2 is encoded as 0b00.

The pre-UAL syntax LDR<c>SB is equivalent to LDRSB<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

LDRSB<c><q> <Rt>, [<Rn>, <Rm>{, LSL #<imm>}]	オフセット：index==TRUE、wback==FALSE
LDRSB<c><q> <Rt>, [<Rn>, +/-<Rm>]	オフセット：index==TRUE、wback==FALSE
LDRSB<c><q> <Rt>, [<Rn>, +/-<Rm>]!	プリインデクス：index==TRUE、wback==TRUE
LDRSB<c><q> <Rt>, [<Rn>], +/-<Rm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rt> デスティネーションレジスタ

<Rn> ベースレジスタ。このレジスタには SP を指定できます。PC は、ARM 命令セットでのみ指定できます。

+/- オプションでシフトされる <Rm> の値をベースレジスタの値に加算する場合 (add == TRUE) は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、add == FALSE）は、「−」にします。

<Rm> アドレス生成のためにオプションとして左にシフトされて、<Rn> の値に加算されるオフセット

<imm> 指定した場合、左シフト量 (1 〜 3) が <Rm> から読み出された値に適用されます。エンコード T2 のみ許可され、<imm> は imm2 にエンコードされます。

省略した場合、シフトは指定されず、すべてのエンコードが許可されます。エンコード T2 では、imm2 は 0b00 としてエンコードされます。

UAL 以前の構文の LDR<c>SB は、LDRSB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.81 LDRSBT

Load Register Signed Byte Unprivileged loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page A8-13.

The memory access is restricted as if the processor were running in User mode. (This makes no difference if the processor is actually running in User mode.)

The Thumb instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The ARM instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

Encoding T1 ARMv6T2, ARMv7

```
LDRSBT<c> <Rt>,[<Rn>,#<imm8>]
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	1 0 0 1	0 0 1 1	Rn	Rt	1 1 1 0	imm8	

```
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

Encoding A1 ARMv6T2, ARMv7

```
LDRSBT<c> <Rt>, [<Rn>] {, #+/-<imm8>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	0	U	1	1	1	Rn	Rt	imm4H	1 1 0 1	imm4L

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

Encoding A2 ARMv6T2, ARMv7

```
LDRSBT<c> <Rt>, [<Rn>], +/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	0	U	0	1	1	Rn	Rt	(0) (0) (0) (0)	1 1 0 1	Rm

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.81 LDRSBT

LDRSBT（非特権レジスタロード符号付きバイト）は、メモリから 1 バイトをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。メモリアクセスの詳細については、*メモリアクセス*P. A8- 13 を参照して下さい。

プロセッサがユーザモードで実行されている場合と同様にメモリアクセスが制限されます（実際にプロセッサがユーザモードで実行されている場合は、何も変わりません）。

Thumb 命令では、オフセット アドレシングモードを使用します。このモードでは、メモリアクセスに使用されるアドレスを、ベースレジスタの値とイミディエートオフセットから計算し、ベースレジスタの値は変更しません。

ARM 命令では、ポストインデクス アドレシングモードを使用します。このモードでは、ベースレジスタの値をメモリアクセス用のアドレスとして使用し、ベースレジスタの値とオフセットから新しいアドレスを計算して、ベースレジスタに書き戻します。オフセットは、イミディエート値またはレジスタの値です。

エンコード T1 ARMv6T2、ARMv7

```
LDRSBT<c> <Rt>,[<Rn>,#<imm8>]
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	1 0 0 1	0 0 1 1	Rn	Rt	1 1 1 0	imm8	

```
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

エンコード A1 ARMv6T2、ARMv7

```
LDRSBT<c> <Rt>, [<Rn>] {, #+/-<imm8>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	0	U	1	1	1	Rn	Rt	imm4H	1 1 0 1	imm4L

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

エンコード A2 ARMv6T2、ARMv7

```
LDRSBT<c> <Rt>, [<Rn>], +/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	0	U	0	1	1	Rn	Rt	(0) (0) (0) (0)	1 1 0 1	Rm

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRSBT<c><q> <Rt>, [<Rn> {, #<imm>}]`	Offset: Thumb only
`LDRSBT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	Post-indexed: ARM only
`LDRSBT<c><q> <Rt>, [<Rn>], +/-<Rm>`	Post-indexed: ARM only

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used.

`+/-` Is + or omitted if `<imm>` or the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, `add == FALSE`).

`<imm>` The immediate offset applied to the value of `<Rn>`. Any value in the range 0-255 is permitted. `<imm>` can be omitted, meaning an offset of 0.

`<Rm>` Contains the offset that is applied to the value of `<Rn>` to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = SignExtend(MemU_unpriv[address,1], 32);
    if postindex then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRSBT<c><q> <Rt>, [<Rn> {, #<imm>}]`	オフセット：Thumb のみ
`LDRSBT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	ポストインデクス：ARM のみ
`LDRSBT<c><q> <Rt>, [<Rn>], +/-<Rm>`	ポストインデクス：ARM のみ

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`+/-` `<imm>` またはオプションとしてシフトされる `<Rm>` の値をベースレジスタの値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、`add == FALSE`）は、「-」にします。

`<imm>` `<Rn>` の値に適用されるイミディエートオフセット。0 ～ 255 の任意の値が許可されます。`<imm>` は省略可能で、その場合オフセットは 0 と見なされます。

`<Rm>` アドレス生成のために `<Rn>` の値に適用されるオフセットを格納します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = SignExtend(MemU_unpriv[address,1], 32);
    if postindex then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.82 LDRSH (immediate)

Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
LDRSH<c> <Rt>,[<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	0	1	1	Rn				Rt				imm12											

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

Encoding T2 ARMv6T2, ARMv7

```
LDRSH<c> <Rt>,[<Rn>,#-<imm8>]
LDRSH<c> <Rt>,[<Rn>],#+/-<imm8>
LDRSH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	1	1	Rn				Rt				1	P	U	W	imm8							

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Unallocated memory hints";
if P == '1' && U == '1' && W == '0' then SEE LDRSHT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRSH<c> <Rt>,[<Rn>{,#+/-<imm8>}]
LDRSH<c> <Rt>,[<Rn>],#+/-<imm8>
LDRSH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	P	U	1	W	1	Rn				Rt				imm4H				1	1	1	1	imm4L			

```
if Rn == '1111' then SEE LDRSH (literal);
if P == '0' && W == '1' then SEE LDRSHT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

Unallocated memory hints See *Load halfword, memory hints* on page A6-26

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.82 LDRSH（イミディエート）

LDRSH（レジスタロード符号付きハーフワード（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
LDRSH<c> <Rt>,[<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	0	1	1	Rn				Rt				imm12											

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

```
LDRSH<c> <Rt>,[<Rn>,#-<imm8>]
LDRSH<c> <Rt>,[<Rn>],#+/-<imm8>
LDRSH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	1	1	Rn				Rt				1	P	U	W	imm8							

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Unallocated memory hints";
if P == '1' && U == '1' && W == '0' then SEE LDRSHT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRSH<c> <Rt>,[<Rn>{,#+/-<imm8>}]
LDRSH<c> <Rt>,[<Rn>],#+/-<imm8>
LDRSH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	P	U	1	W	1	Rn				Rt				imm4H				1	1	1	1	imm4L			

```
if Rn == '1111' then SEE LDRSH (literal);
if P == '0' && W == '1' then SEE LDRSHT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

未割り当てメモリヒント *ハーフワードロード、メモリヒント* P. A6-26 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRSH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRSH<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRSH<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used. For PC use see *LDRSH (literal)* on page A8-170.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` The immediate offset used to form the address, Values are 0-4095 for encoding T1, and 0-255 for encoding T2 or A1. For the offset syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `LDR<c>SH` is equivalent to `LDRSH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = SignExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRSH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDRSH<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDRSH<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC の指定については、*LDRSH（リテラル）* P. A8-170 を参照して下さい。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。値の範囲は、エンコード T1 では 0 ～ 4095、エンコード T2 または A1 では 0 ～ 255 です。オフセット構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の `LDR<c>SH` は、`LDRSH<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = SignExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.83 LDRSH (literal)

Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
LDRSH<c> <Rt>,<label>
LDRSH<c> <Rt>,[PC,#-0]                                  Special case
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	U	0	1	1	1	1	1	1	Rt				imm12											

```
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRSH<c> <Rt>,<label>
LDRSH<c> <Rt>,[PC,#-0]                                  Special case
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	(1)	U	1	(0)	1	1	1	1	1	Rt				imm4H				1	1	1	1	imm4L			

```
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);  add = (U == '1');
if t == 15 then UNPREDICTABLE;
```

Unallocated memory hints See *Load halfword, memory hints* on page A6-26

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.83 LDRSH（リテラル）

LDRSH（レジスタロード符号付きハーフワード（リテラル））は、PC の値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
LDRSH<c> <Rt>,<label>
LDRSH<c> <Rt>,[PC,#-0]                                  特殊なケース
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	U	0	1	1	1	1	1	1	Rt				imm12											

```
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRSH<c> <Rt>,<label>
LDRSH<c> <Rt>,[PC,#-0]                                  特殊なケース
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	(1)	U	1	(0)	1	1	1	1	1	Rt				imm4H				1	1	1	1	imm4L			

```
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);  add = (U == '1');
if t == 15 then UNPREDICTABLE;
```

未割り当てメモリヒント *ハーフワードロード、メモリヒント* P. A6-26 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRSH<c><q> <Rt>, <label>`	Normal form
`LDRSH<c><q> <Rt>, [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC,4)` value of the `ADR` instruction to this label. Permitted values of the offset are:

Encoding T1 any value in the range -4095 to 4095

Encoding A1 any value in the range -255 to 255.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

The pre-UAL syntax `LDR<c>SH` is equivalent to `LDRSH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(15);
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    if UnalignedSupport() || address<0> = '0' then
        R[t] = SignExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRSH<c><q> <Rt>, <label>`	通常の形式
`LDRSH<c><q> <Rt>, [PC, #+/-<imm>]`	代替形式

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<label>` `<Rt>` にロードされるリテラルデータ項目のラベル。アセンブラは、`ADR` 命令の `Align(PC,4)` の値からこのラベルまでのオフセットに必要な値を計算します。オフセットに許容される値の範囲は次のとおりです。

エンコード T1 − 4095 〜 4095 の任意の値

エンコード A1 − 255 〜 255 の任意の値

オフセットが 0 または正の場合、`imm32` はオフセットと等しく、`add == TRUE` になります。

オフセットが負の場合、`imm32` は負のオフセットと等しく、`add == FALSE` になります。

代替構文では、通常の構文では指定できない 0 の減算など、オフセットおよびイミディエートオフセットの加算または減算を個別に指定することが許可されます。詳細については、*UAL 命令構文でのラベルの使用* P. A4-5 を参照して下さい。

UAL 以前の構文の `LDR<c>SH` は、`LDRSH<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(15);
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    if UnalignedSupport() || address<0> = '0' then
        R[t] = SignExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.84 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LDRSH<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	1 1 1	Rm	Rn	Rt

```
if CurrentInstrSet() == InstrSet_ThumbEE then SEE "Modified operation in ThumbEE";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
LDRSH<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11	10 9 8 7 6	5 4 3 2 1 0
1 1 1 1	1 1	0	1	0	1	1	Rn	Rt	0	0 0 0 0 0	imm2 Rm

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRSH<c> <Rt>,[<Rn>,+/-<Rm>]{!}
LDRSH<c> <Rt>,[<Rn>],+/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	0	W	1	Rn	Rt	(0) (0) (0) (0)	1 1 1 1	Rm

```
if P == '0' && W == '1' then SEE LDRSHT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

Unallocated memory hints See *Load halfword, memory hints* on page A6-26

Modified operation in ThumbEE See *LDRSH (register)* on page A9-11

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.84 LDRSH（レジスタ）

LDRSH（レジスタロード符号付きハーフワード（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。オフセットレジスタの値は、左に 0、1、2、3 ビットシフトできます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LDRSH<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	1 1 1	Rm	Rn	Rt

```
if CurrentInstrSet() == InstrSet_ThumbEE then SEE "Modified operation in ThumbEE";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
LDRSH<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11	10 9 8 7 6	5 4 3 2 1 0
1 1 1 1	1 1	0	1	0	1	1	Rn	Rt	0	0 0 0 0 0	imm2 Rm

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRSH<c> <Rt>,[<Rn>,+/-<Rm>]{!}
LDRSH<c> <Rt>,[<Rn>],+/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	0	W	1	Rn	Rt	(0) (0) (0) (0)	1 1 1 1	Rm

```
if P == '0' && W == '1' then SEE LDRSHT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

未割り当てメモリヒント *ハーフワードロード、メモリヒント* P. A6- 26 を参照して下さい。

ThumbEE で変更された動作 *LDRSH（レジスタ）* P. A9- 11 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRSH<c><q> <Rt>, [<Rn>, <Rm>{, LSL #<imm>}]`	Offset: `index==TRUE, wback==FALSE`
`LDRSH<c><q> <Rt>, [<Rn>, +/-<Rm>]`	Offset: `index==TRUE, wback==FALSE`
`LDRSH<c><q> <Rt>, [<Rn>, +/-<Rm>]!`	Pre-indexed: `index==TRUE, wback==TRUE`
`LDRSH<c><q> <Rt>, [<Rn>], +/-<Rm>`	Post-indexed: `index==FALSE, wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used. The PC can be used only in the ARM instruction set.

`+/-` Is + or omitted if the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, add == FALSE).

`<Rm>` Contains the offset that is optionally left shifted and added to the value of `<Rn>` to form the address.

`<imm>` If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. Only encoding T2 is permitted, and `<imm>` is encoded in imm2.

If absent, no shift is specified and all encodings are permitted. In encoding T2, imm2 is encoded as `0b00`.

The pre-UAL syntax `LDR<c>SH` is equivalent to `LDRSH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = SignExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRSH<c><q> <Rt>, [<Rn>, <Rm>{, LSL #<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDRSH<c><q> <Rt>, [<Rn>, +/-<Rm>]`	オフセット：`index==TRUE`、`wback==FALSE`
`LDRSH<c><q> <Rt>, [<Rn>, +/-<Rm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`LDRSH<c><q> <Rt>, [<Rn>], +/-<Rm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC は、ARM 命令セットでのみ指定できます。

`+/-` オプションでシフトされる `<Rm>` の値をベースレジスタの値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、add == FALSE）は、「−」にします。

`<Rm>` アドレス生成のためにオプションとして左にシフトされて、`<Rn>` の値に加算されるオフセット

`<imm>` 指定した場合、左シフト量 (1 〜 3) が `<Rm>` から読み出された値に適用されます。エンコード T2 のみ許可され、`<imm>` は imm2 にエンコードされます。

省略した場合、シフトは指定されず、すべてのエンコードが許可されます。エンコード T2 では、imm2 は `0b00` としてエンコードされます。

UAL 以前の構文の `LDR<c>SH` は、`LDRSH<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = SignExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.85 LDRSHT

Load Register Signed Halfword Unprivileged loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses* on page A8-13.

The memory access is restricted as if the processor were running in User mode. (This makes no difference if the processor is actually running in User mode.)

The Thumb instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The ARM instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

Encoding T1 ARMv6T2, ARMv7

```
LDRSHT<c> <Rt>,[<Rn>,#<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	1	1	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then SEE LDRSH (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

Encoding A1 ARMv6T2, ARMv7

```
LDRSHT<c> <Rt>, [<Rn>] {, #+/-<imm8>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	U	1	1	1	Rn				Rt				imm4H				1	1	1	1	imm4L			

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

Encoding A2 ARMv6T2, ARMv7

```
LDRSHT<c> <Rt>, [<Rn>], +/-<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	U	0	1	1	Rn				Rt				(0)	(0)	(0)	(0)	1	1	1	1	Rm			

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.85 LDRSHT

LDRSHT（非特権レジスタロード符号付きハーフワード）は、メモリから 1 ハーフワードをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

プロセッサがユーザモードで実行されている場合と同様にメモリアクセスが制限されます（実際にプロセッサがユーザモードで実行されている場合は、何も変わりません）。

Thumb 命令では、オフセット アドレシングモードを使用します。このモードでは、メモリアクセスに使用されるアドレスを、ベースレジスタの値とイミディエートオフセットから計算し、ベースレジスタの値は変更しません。

ARM 命令では、ポストインデクス アドレシングモードを使用します。このモードでは、ベースレジスタの値をメモリアクセス用のアドレスとして使用し、ベースレジスタの値とオフセットから新しいアドレスを計算して、ベースレジスタに書き戻します。オフセットは、イミディエート値またはレジスタの値です。

エンコード T1 ARMv6T2、ARMv7

```
LDRSHT<c> <Rt>,[<Rn>,#<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	1	1	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then SEE LDRSH (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

エンコード A1 ARMv6T2、ARMv7

```
LDRSHT<c> <Rt>, [<Rn>] {, #+/-<imm8>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	U	1	1	1	Rn				Rt				imm4H				1	1	1	1	imm4L			

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

エンコード A2 ARMv6T2、ARMv7

```
LDRSHT<c> <Rt>, [<Rn>], +/-<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	U	0	1	1	Rn				Rt				(0)	(0)	(0)	(0)	1	1	1	1	Rm			

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRSHT<c><q> <Rt>, [<Rn> {, #<imm>}]`	Offset: Thumb only
`LDRSHT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	Post-indexed: ARM only
`LDRSHT<c><q> <Rt>, [<Rn>], +/-<Rm>`	Post-indexed: ARM only

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used.

`+/-` Is + or omitted if `<imm>` or the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, `add == FALSE`).

`<imm>` The immediate offset applied to the value of `<Rn>`. Any value in the range 0-255 is permitted. `<imm>` can be omitted, meaning an offset of 0.

`<Rm>` Contains the offset that is applied to the value of `<Rn>` to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = SignExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRSHT<c><q> <Rt>, [<Rn> {, #<imm>}]`	オフセット：Thumb のみ
`LDRSHT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	ポストインデクス：ARM のみ
`LDRSHT<c><q> <Rt>, [<Rn>], +/-<Rm>`	ポストインデクス：ARM のみ

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`+/-` `<imm>` またはオプションとしてシフトされる `<Rm>` の値をベースレジスタの値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、`add == FALSE`）は、「-」にします。

`<imm>` `<Rn>` の値に適用されるイミディエートオフセット。0 ～ 255 の任意の値が許可されます。`<imm>` は省略可能で、その場合オフセットは 0 と見なされます。

`<Rm>` アドレス生成のために `<Rn>` の値に適用されるオフセットを格納します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    if UnalignedSupport() || address<0> = '0' then
        R[t] = SignExtend(data, 32);
    else // Can only apply before ARMv7
        R[t] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.86 LDRT

Load Register Unprivileged loads a word from memory, and writes it to a register. For information about memory accesses see *Memory accesses* on page A8-13.

The memory access is restricted as if the processor were running in User mode. (This makes no difference if the processor is actually running in User mode.)

The Thumb instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The ARM instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

Encoding T1 ARMv6T2, ARMv7

```
LDRT<c> <Rt>,[<Rn>,#<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	1	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRT<c> <Rt>, [<Rn>] {, #+/-<imm12>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	0	0	U	0	1	1	Rn				Rt				imm12											

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

Encoding A2 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDRT<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	U	0	1	1	Rn				Rt				imm5					type		0	Rm			

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && m == n then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.86 LDRT

LDRT（非特権レジスタロード）は、メモリから 1 ワードをロードし、レジスタに書き込みます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

プロセッサがユーザモードで実行されている場合と同様にメモリアクセスが制限されます（実際にプロセッサがユーザモードで実行されている場合は、何も変わりません）。

Thumb 命令では、オフセット アドレシングモードを使用します。このモードでは、メモリアクセスに使用されるアドレスを、ベースレジスタの値とイミディエートオフセットから計算し、ベースレジスタの値は変更しません。

ARM 命令では、ポストインデクス アドレシングモードを使用します。このモードでは、ベースレジスタの値をメモリアクセス用のアドレスとして使用し、ベースレジスタの値とオフセットから新しいアドレスを計算して、ベースレジスタに書き戻します。オフセットは、イミディエート値またはオプションとしてシフトされるレジスタの値です。

エンコード T1 ARMv6T2、ARMv7

```
LDRT<c> <Rt>,[<Rn>,#<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	1	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRT<c> <Rt>, [<Rn>] {, #+/-<imm12>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	0	0	U	0	1	1	Rn				Rt				imm12											

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

エンコード A2 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDRT<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	U	0	1	1	Rn				Rt				imm5					type		0	Rm			

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && m == n then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`LDRT<c><q> <Rt>, [<Rn> {, #<imm>}]`	Offset: Thumb only
`LDRT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	Post-indexed: ARM only
`LDRT<c><q> <Rt>, [<Rn>], +/-<Rm> {, <shift>}`	Post-indexed: ARM only

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. The SP can be used.

`+/-` Is + or omitted if `<imm>` or the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, `add == FALSE`).

`<imm>` The immediate offset applied to the value of `<Rn>`. Values are 0-255 for encoding T1, and 0-4095 for encoding A1. `<imm>` can be omitted, meaning an offset of 0.

`<Rm>` Contains the offset that is optionally shifted and applied to the value of `<Rn>` to form the address.

`<shift>` The shift to apply to the value read from `<Rm>`. If omitted, no shift is applied. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

The pre-UAL syntax `LDR<c>T` is equivalent to `LDRT<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then Shift(R[m], shift_t, shift_n, APSR.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,4];
    if postindex then R[n] = offset_addr;
    if t == 15 then  // Only possible for encodings A1 and A2
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    elsif UnalignedSupport() || address<1:0> = '00' then
        R[t] = data;
    else // Can only apply before ARMv7
        if CurrentInstrSet() == InstrSet_ARM then
            R[t] = ROR(data, 8*UInt(address<1:0>));
        else
            R[t] = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`LDRT<c><q> <Rt>, [<Rn> {, #<imm>}]`	オフセット：Thumb のみ
`LDRT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	ポストインデクス：ARM のみ
`LDRT<c><q> <Rt>, [<Rn>], +/-<Rm> {, <shift>}`	ポストインデクス：ARM のみ

各項目の説明については以下を参照して下さい。

`<c><q>`. *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>`. デスティネーションレジスタ

`<Rn>`. ベースレジスタ。このレジスタには SP を指定できます。

`+/-`. `<imm>` またはオプションとしてシフトされる `<Rm>` の値をベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、`add == FALSE`）は、「-」にします。

`<imm>`. `<Rn>` の値に適用されるイミディエートオフセット。値の範囲は、エンコード T1 では 0 〜 255、エンコード A1 では 0 〜 4095 です。`<imm>` は省略可能で、その場合オフセットは 0 と見なされます。

`<Rm>`. アドレス生成のためにオプションとしてシフトされ、`<Rn>` の値に適用されるオフセットを格納します。

`<shift>`. `<Rm>` から読み出された値に適用されるシフト。省略した場合、シフトは適用されません。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8- 10 を参照して下さい。

UAL 以前の構文の `LDR<c>T` は、`LDRT<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then Shift(R[m], shift_t, shift_n, APSR.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,4];
    if postindex then R[n] = offset_addr;
    if t == 15 then  // Only possible for encodings A1 and A2
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    elsif UnalignedSupport() || address<1:0> = '00' then
        R[t] = data;
    else // Can only apply before ARMv7
        if CurrentInstrSet() == InstrSet_ARM then
            R[t] = ROR(data, 8*UInt(address<1:0>));
        else
            R[t] = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.87 LEAVEX

LEAVEX causes a change from ThumbEE to Thumb state, or has no effect in Thumb state. For details see *ENTERX, LEAVEX* on page A9-7.

A8.6.88 LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LSLS <Rd>,<Rm>,#<imm5>                    Outside IT block.
LSL<c> <Rd>,<Rm>,#<imm5>                  Inside IT block.
```

15 14 13	12 11	10 9 8 7 6	5 4 3	2 1 0
0 0 0	0 0	imm5	Rm	Rd

```
if imm5 == '00000' then SEE MOV (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('00', imm5);
```

Encoding T2 ARMv6T2, ARMv7

```
LSL{S}<c>.W <Rd>,<Rm>,#<imm5>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15	14 13 12 11	10 9 8 7	6 5	4 3 2 1 0
1 1 1 0	1 0 1	0 0 1 0	S 1 1 1 1	(0)	imm3	Rd	imm2	0 0 Rm

```
if (imm3:imm2) == '00000' then SEE MOV (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('00', imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LSL{S}<c> <Rd>,<Rm>,#<imm5>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5 4	3 2 1 0
cond	0 0	0	1 1 0 1	S	(0) (0) (0) (0)	Rd	imm5	0 0 0	Rm

```
if imm5 == '00000' then SEE MOV (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('00', imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.87 LEAVEX

LEAVEX は ThumbEE 状態から Thumb 状態への変更を引き起こしますが、Thumb 状態では影響を与えません。詳細については、*ENTERX*、*LEAVEX* P. A9-7 を参照して下さい。

A8.6.88 LSL（イミディエート）

LSL（論理左シフト（イミディエート））は、レジスタの値のビットをイミディエート値で指定されたビット数だけ左にシフト（空いたビットには 0 が入ります）して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LSLS <Rd>,<Rm>,#<imm5>                    IT ブロック外部
LSL<c> <Rd>,<Rm>,#<imm5>                  IT ブロック内部
```

15 14 13	12 11	10 9 8 7 6	5 4 3	2 1 0
0 0 0	0 0	imm5	Rm	Rd

```
if imm5 == '00000' then SEE MOV (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('00', imm5);
```

エンコード T2 ARMv6T2、ARMv7

```
LSL{S}<c>.W <Rd>,<Rm>,#<imm5>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15	14 13 12 11	10 9 8 7	6 5	4 3 2 1 0
1 1 1 0	1 0 1	0 0 1 0	S 1 1 1 1	(0)	imm3	Rd	imm2	0 0 Rm

```
if (imm3:imm2) == '00000' then SEE MOV (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('00', imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LSL{S}<c> <Rd>,<Rm>,#<imm5>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5 4	3 2 1 0
cond	0 0	0	1 1 0 1	S	(0) (0) (0) (0)	Rd	imm5	0 0 0	Rm

```
if imm5 == '00000' then SEE MOV (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('00', imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
LSL{S}<c><q>  {<Rd>,} <Rm>, #<imm5>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rm>	The first operand register.
<imm5>	The shift amount, in the range 1 to 31. See *Shifts applied to a register* on page A8-10.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_LSL, shift_n, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
LSL{S}<c><q>  {<Rd>,} <Rm>, #<imm5>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rm>	最初のオペランドレジスタ
<imm5>	1 ～ 31 の範囲のシフト量。詳細については、*レジスタに適用されるシフト* P. A8- 10 を参照して下さい。

動作


```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_LSL, shift_n, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```


例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.89 LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LSLS    <Rdn>,<Rm>                    Outside IT block.
LSL<c> <Rdn>,<Rm>                     Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

Encoding T2 ARMv6T2, ARMv7

```
LSL{S}<c>.W <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	S	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LSL{S}<c> <Rd>,<Rn>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				Rm				0	0	0	1	Rn			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.89 LSL（レジスタ）

LSL（論理左シフト（レジスタ））は、レジスタの値のビットを可変のビット数だけ左にシフト（空いたビットには 0 が入ります）して、結果をデスティネーションレジスタに書き込みます。ローテートするビット数は、レジスタの最下位バイトから読み出されます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LSLS    <Rdn>,<Rm>                    IT ブロック外部
LSL<c> <Rdn>,<Rm>                     IT ブロック内部
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

エンコード T2 ARMv6T2、ARMv7

```
LSL{S}<c>.W <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	S	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LSL{S}<c> <Rd>,<Rn>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				Rm				0	0	0	1	Rn			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
LSL{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_LSL, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
LSL{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	最下位バイトにシフトのビット数を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_LSL, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.90 LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LSRS <Rd>,<Rm>,#<imm>
LSR<c> <Rd>,<Rm>,#<imm>
```

Outside IT block.
Inside IT block.

15 14 13	12 11	10 9 8 7 6	5 4 3	2 1 0
0 0 0	0 1	imm5	Rm	Rd

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('01', imm5);
```

Encoding T2 ARMv6T2, ARMv7

```
LSR{S}<c>.W <Rd>,<Rm>,#<imm>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 0 1 0	S 1 1 1 1	(0) imm3	Rd	imm2	0 1	Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('01', imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LSR{S}<c> <Rd>,<Rm>,#<imm>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5 4	3 2 1 0
cond	0 0	0	1 1 0 1	S	(0) (0) (0) (0)	Rd	imm5	0 1 0	Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('01', imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.90 LSR（イミディエート）

LSR（論理右シフト（イミディエート））は、レジスタの値のビットをイミディエート値で指定されたビット数だけ右にシフト（空いたビットには 0 が入ります）して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LSRS <Rd>,<Rm>,#<imm>
LSR<c> <Rd>,<Rm>,#<imm>
```

IT ブロック外部
IT ブロック内部

15 14 13	12 11	10 9 8 7 6	5 4 3	2 1 0
0 0 0	0 1	imm5	Rm	Rd

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('01', imm5);
```

エンコード T2 ARMv6T2、ARMv7

```
LSR{S}<c>.W <Rd>,<Rm>,#<imm>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 0 1 0	S 1 1 1 1	(0) imm3	Rd	imm2	0 1	Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('01', imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LSR{S}<c> <Rd>,<Rm>,#<imm>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5 4	3 2 1 0
cond	0 0	0	1 1 0 1	S	(0) (0) (0) (0)	Rd	imm5	0 1 0	Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('01', imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
LSR{S}<c><q>  {<Rd>,} <Rm>, #<imm>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rm>	The first operand register.
<imm>	The shift amount, in the range 1 to 32. See *Shifts applied to a register* on page A8-10.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_LSR, shift_n, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
LSR{S}<c><q>  {<Rd>,} <Rm>, #<imm>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rm>	最初のオペランドレジスタ
<imm>	1 ～ 32 の範囲のシフト量。詳細については、*レジスタに適用されるシフト* P. A8- 10 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_LSR, shift_n, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.91 LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
LSRS   <Rdn>,<Rm>                    Outside IT block.
LSR<c> <Rdn>,<Rm>                    Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

Encoding T2 ARMv6T2, ARMv7

```
LSR{S}<c>.W <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	1	S	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LSR{S}<c> <Rd>,<Rn>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				Rm				0	0	1	1	Rn			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.91 LSR（レジスタ）

LSR（論理右シフト（レジスタ））は、レジスタの値のビットを可変のビット数だけ右にシフト（空いたビットには 0 が入ります）して、結果をデスティネーションレジスタに書き込みます。ローテートするビット数は、レジスタの最下位バイトから読み出されます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
LSRS   <Rdn>,<Rm>                    IT ブロック外部
LSR<c> <Rdn>,<Rm>                    IT ブロック内部
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

エンコード T2 ARMv6T2、ARMv7

```
LSR{S}<c>.W <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	1	S	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LSR{S}<c> <Rd>,<Rn>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				Rm				0	0	1	1	Rn			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
LSR{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_LSR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
LSR{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	最下位バイトにシフトのビット数を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_LSR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.92 MCR, MCR2

Move to Coprocessor from ARM core register passes the value of an ARM core register to a coprocessor. If no coprocessor can execute the instruction, an Undefined Instruction exception is generated.

This is a generic coprocessor instruction. Some of the fields have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer. These fields are the opc1, opc2, CRn, and CRm fields.

For more information about the coprocessors see *Coprocessor support* on page A2-68.

Encoding T1 / A1 ARMv6T2, ARMv7 for encoding T1
ARMv4*, ARMv5T*, ARMv6*, ARMv7 for encoding A1

```
MCR<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}
```

15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 0	1 1 1 0	opc1	0	CRn	Rt	coproc	opc2	1	CRm

31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
cond	1 1 1 0	opc1	0	CRn	Rt	coproc	opc2	1	CRm

```
if coproc == '101x' then SEE "Advanced SIMD and VFP";
t = UInt(Rt);  cp = UInt(coproc);
if t == 15 || (t == 13 && (CurrentInstrSet() != InstrSet_ARM)) then UNPREDICTABLE;
```

Encoding T2 / A2 ARMv6T2, ARMv7 for encoding T2
ARMv5T*, ARMv6*, ARMv7 for encodingA2

```
MCR2<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}
```

15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1	1 1 1 0	opc1	0	CRn	Rt	coproc	opc2	1	CRm

31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1	1 1 1 0	opc1	0	CRn	Rt	coproc	opc2	1	CRm

```
t = UInt(Rt);  cp = UInt(coproc);
if t == 15 || (t == 13 && (CurrentInstrSet() != InstrSet_ARM)) then UNPREDICTABLE;
```

Advanced SIMD and VFP See *8, 16, and 32-bit transfer between ARM core and extension registers* on page A7-31

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.92 MCR、MCR2

MCR、MCR2（ARM コアレジスタからコプロセッサへの移動）は、ARM コアレジスタの値をコプロセッサに転送します。コプロセッサが命令を実行できない場合は、未定義命令例外が発生します。

これは、汎用のコプロセッサ命令です。一部のフィールドの機能はアーキテクチャで定義されていないため、コプロセッサ命令セットの設計者が自由に使用することができます。これらのフィールドは、opc1、opc2、CRn、CRm です。

コプロセッサの詳細については、*コプロセッサのサポート* P. A2-68 を参照して下さい。

エンコード T1/A1 ARMv6T2、ARMv7（エンコード T1 の場合）
ARMv4*、ARMv5T*、ARMv6*、ARMv7（エンコード A1 の場合）

```
MCR<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}
```

15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 0	1 1 1 0	opc1	0	CRn	Rt	coproc	opc2	1	CRm

31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
cond	1 1 1 0	opc1	0	CRn	Rt	coproc	opc2	1	CRm

```
if coproc == '101x' then SEE "Advanced SIMD and VFP";
t = UInt(Rt);  cp = UInt(coproc);
if t == 15 || (t == 13 && (CurrentInstrSet() != InstrSet_ARM)) then UNPREDICTABLE;
```

エンコード T2/A2 ARMv6T2、ARMv7（エンコード T2 の場合）
ARMv5T*、ARMv6*、ARMv7（エンコード A2 の場合）

```
MCR2<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}
```

15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1	1 1 1 0	opc1	0	CRn	Rt	coproc	opc2	1	CRm

31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1	1 1 1 0	opc1	0	CRn	Rt	coproc	opc2	1	CRm

```
t = UInt(Rt);  cp = UInt(coproc);
if t == 15 || (t == 13 && (CurrentInstrSet() != InstrSet_ARM)) then UNPREDICTABLE;
```

アドバンスト SIMD および VFP *ARM コアレジスタと拡張機能レジスタとの 8、16、32 ビット転送* P. A7-31 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MCR{2}<c><q>  <coproc>, #<opc1>, <Rt>, <CRn>, <CRm>{, #<opc2>}
```

where:

2	If specified, selects encoding T2 / A2. If omitted, selects encoding T1 / A1.
<c><q>	See *Standard assembler syntax fields* on page A8-7. An ARM MCR2 instruction must be unconditional.
<coproc>	The name of the coprocessor. The standard generic coprocessor names are p0, p1, …, p15.
<opc1>	Is a coprocessor-specific opcode in the range 0 to 7.
<Rt>	Is the ARM core register whose value is transferred to the coprocessor.
<CRn>	Is the destination coprocessor register.
<CRm>	Is an additional destination coprocessor register.
<opc2>	Is a coprocessor-specific opcode in the range 0-7. If omitted, <opc2> is assumed to be 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_SendOneWord(R[t], cp, ThisInstr());
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MCR{2}<c><q>  <coproc>, #<opc1>, <Rt>, <CRn>, <CRm>{, #<opc2>}
```

各項目の説明については以下を参照して下さい。

2	指定した場合、エンコード T2/A2 が選択されます。省略した場合、エンコード T1/A1 が選択されます。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM MCR2 は常に無条件命令です。
<coproc>	コプロセッサの名前。標準の汎用コプロセッサ名は、p0、p1、...、p15 です。
<opc1>	コプロセッサ固有のオペコードで、範囲は 0 ～ 7 です。
<Rt>	値をコプロセッサに転送する ARM コアレジスタ
<CRn>	デスティネーション コプロセッサレジスタ
<CRm>	追加のデスティネーション コプロセッサレジスタ
<opc2>	コプロセッサ固有のオペコードで、範囲は 0 ～ 7 です。省略した場合、<opc2> が 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_SendOneWord(R[t], cp, ThisInstr());
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.93 MCRR, MCRR2

Move to Coprocessor from two ARM core registers passes the values of two ARM core registers to a coprocessor. If no coprocessor can execute the instruction, an Undefined Instruction exception is generated.

This is a generic coprocessor instruction. The opc1 and CRm fields have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer.

For more information about the coprocessors see *Coprocessor support* on page A2-68.

Encoding T1 / A1 ARMv6T2, ARMv7 for encoding T1
ARMv5TE*, ARMv6*, ARMv7 for encoding A1

```
MCRR<c> <coproc>,<opc1>,<Rt>,<Rt2>,<CRm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	1	0	0	Rt2				Rt				coproc				opc1				CRm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	0	0	1	0	0	Rt2				Rt				coproc				opc1				CRm			

```
if coproc == '101x' then SEE "Advanced SIMD and VFP";
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if (t == 13 || t2 == 13) && (CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

Encoding T2 / A2 ARMv6T2, ARMv7 for encoding T2
ARMv6*, ARMv7 for encoding A2

```
MCRR2<c> <coproc>,<opc1>,<Rt>,<Rt2>,<CRm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	1	0	0	Rt2				Rt				coproc				opc1				CRm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	1	0	0	Rt2				Rt				coproc				opc1				CRm			

```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if (t == 13 || t2 == 13) && (CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

Advanced SIMD and VFP See *64-bit transfers between ARM core and extension registers* on page A7-32

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.93 MCRR、MCRR2

MCRR、MCRR2（2 つの ARM コアレジスタからコプロセッサへの移動）は、2 つの ARM コアレジスタの値をコプロセッサに転送します。コプロセッサが命令を実行できない場合は、未定義命令例外が発生します。

これは、汎用のコプロセッサ命令です。opc1 フィールドと CRm フィールドの機能はアーキテクチャで定義されていないため、コプロセッサ命令セットの設計者が自由に使用することができます。

コプロセッサの詳細については、*コプロセッサのサポート* P. A2-68 を参照して下さい。

エンコード T1/A1 ARMv6T2、ARMv7（エンコード T1 の場合）
ARMv5TE*、ARMv6*、ARMv7（エンコード A1 の場合）

```
MCRR<c> <coproc>,<opc1>,<Rt>,<Rt2>,<CRm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	1	0	0	Rt2				Rt				coproc				opc1				CRm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	0	0	1	0	0	Rt2				Rt				coproc				opc1				CRm			

```
if coproc == '101x' then SEE "Advanced SIMD and VFP";
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if (t == 13 || t2 == 13) && (CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

エンコード T2/A2 ARMv6T2、ARMv7（エンコード T2 の場合）
ARMv6*、ARMv7（エンコード A2 の場合）

```
MCRR2<c> <coproc>,<opc1>,<Rt>,<Rt2>,<CRm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	1	0	0	Rt2				Rt				coproc				opc1				CRm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	1	0	0	Rt2				Rt				coproc				opc1				CRm			

```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if (t == 13 || t2 == 13) && (CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

アドバンスト SIMD および VFP *ARM コアレジスタと拡張機能レジスタとの 64 ビット転送* P. A7-32 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MCRR{2}<c><q>  <coproc>, #<opc1>, <Rt>, <Rt2>, <CRm>
```

where:

2	If specified, selects encoding T2 / A2. If omitted, selects encoding T1 / A1.
<c><q>	See *Standard assembler syntax fields* on page A8-7. An ARM MCRR2 instruction must be unconditional.
<coproc>	The name of the coprocessor. The standard generic coprocessor names are p0, p1, …, p15.
<opc1>	Is a coprocessor-specific opcode in the range 0 to 15.
<Rt>	Is the first ARM core register whose value is transferred to the coprocessor.
<Rt2>	Is the second ARM core register whose value is transferred to the coprocessor.
<CRm>	Is the destination coprocessor register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_SendTwoWords(R[t], R[t2], cp, ThisInstr());
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MCRR{2}<c><q>  <coproc>, #<opc1>, <Rt>, <Rt2>, <CRm>
```

各項目の説明については以下を参照して下さい。

2	指定した場合、エンコード T2/A2 が選択されます。省略した場合、エンコード T1/A1 が選択されます。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM MCRR2 は常に無条件命令です。
<coproc>	コプロセッサの名前 標準の汎用コプロセッサ名は、p0、p1、…、p15 です。
<opc1>	コプロセッサ固有のオペコードで、範囲は 0 ～ 15 です。
<Rt>	値をコプロセッサに転送する最初の ARM コアレジスタ
<Rt2>	値をコプロセッサに転送する 2 番目の ARM コアレジスタ
<CRm>	デスティネーション コプロセッサレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_SendTwoWords(R[t], R[t2], cp, ThisInstr());
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.94 MLA

Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

In ARM code, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many processor implementations.

Encoding T1 ARMv6T2, ARMv7

```
MLA<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 0 0	Rn	Ra	Rd	0 0 0 0	Rm

```
if Ra == '1111' then SEE MUL;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = FALSE;
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MLA{S}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 0 0 1	S	Rd	Ra	Rm	1 0 0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && d == n then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.94 MLA

MLA（積和演算）は、2 つのレジスタの値を乗算して、3 番目のレジスタの値を加算します。結果の下位 32 ビットがデスティネーションレジスタに書き込まれます。この 32 ビットは、ソースレジスタの値が符号付きの値または符号なしの値のどちらであると見なされるかに依存しません。

ARM コードでは、オプションとして、結果に基づいて条件フラグを更新することができます。このオプションを使用すると、多くのプロセッサ実装で性能が低下します。

エンコード T1 ARMv6T2、ARMv7

```
MLA<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 0 0	Rn	Ra	Rd	0 0 0 0	Rm

```
if Ra == '1111' then SEE MUL;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = FALSE;
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MLA{S}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 0 0 1	S	Rd	Ra	Rm	1 0 0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && d == n then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MLA{S}<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
S can be specified only for the ARM instruction set.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The second operand register.

<Ra> — The register containing the accumulate value.

The pre-UAL syntax MLA<c>S is equivalent to MLAS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = operand1 * operand2 + addend;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
        // else APSR.C unchanged
        // APSR.V always unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MLA{S}<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
S は、ARM 命令セットでのみ指定できます。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — 2 番目のオペランドレジスタ

<Ra> — 累積値を格納しているレジスタ

UAL 以前の構文の MLA<c>S は、MLAS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = operand1 * operand2 + addend;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
        // else APSR.C unchanged
        // APSR.V always unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.95 MLS

Multiply and Subtract multiplies two register values, and subtracts the product from a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

Encoding T1 ARMv6T2, ARMv7

```
MLS<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	0 1 1 0	0 0 0	Rn	Ra	Rd	0 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if BadReg(d) || BadReg(n) || BadReg(m) || BadReg(a) then UNPREDICTABLE;
```

Encoding A1 ARMv6T2, ARMv7

```
MLS<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0	0 1 1 0	Rd	Ra	Rm	1 0 0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.95 MLS

MLS（積減算）は、2 つのレジスタの値を乗算し、その積を 3 番目のレジスタの値から減算します。結果の下位 32 ビットがデスティネーションレジスタに書き込まれます。この 32 ビットは、ソースレジスタの値が符号付きの値または符号なしの値のどちらであると見なされるかに依存しません。

エンコード T1 ARMv6T2、ARMv7

```
MLS<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	0 1 1 0	0 0 0	Rn	Ra	Rd	0 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if BadReg(d) || BadReg(n) || BadReg(m) || BadReg(a) then UNPREDICTABLE;
```

エンコード A1 ARMv6T2、ARMv7

```
MLS<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0	0 1 1 0	Rd	Ra	Rm	1 0 0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MLS<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.
`<Ra>`	The register containing the accumulate value.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = addend - operand1 * operand2;
    R[d] = result<31:0>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MLS<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ
`<Ra>`	累積値を格納しているレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = addend - operand1 * operand2;
    R[d] = result<31:0>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.96 MOV (immediate)

Move (immediate) writes an immediate value to the destination register. It can optionally update the condition flags based on the value.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
MOVS <Rd>,#<imm8>                    Outside IT block.
MOV<c> <Rd>,#<imm8>                  Inside IT block.
```

15 14 13	12 11	10 9 8	7 6 5 4 3 2 1 0
0 0 1	0 0	Rd	imm8

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

Encoding T2 ARMv6T2, ARMv7

```
MOV{S}<c>.W <Rd>,#<const>
```

15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	0	0 0 1 0	S	1 1 1 1	0	imm3	Rd	imm8

```
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

```
MOVW<c> <Rd>,#<imm16>
```

15 14 13 12 11	10	9	8	7	6	5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	1	0	0	1	0	0	imm4	0	imm3	Rd	imm8

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
if BadReg(d) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MOV{S}<c> <Rd>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	1 1 0 1	S	(0) (0) (0) (0)	Rd	imm12

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```

Encoding A2 ARMv6T2, ARMv7

```
MOVW<c> <Rd>,#<imm16>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0 1 1	0 0 0 0	imm4	Rd	imm12

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:imm12, 32);
if d == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.96 MOV（イミディエート）

MOV（移動（イミディエート））は、イミディエート値をデスティネーションレジスタに書き込みます。オプションとして、その値に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
MOVS <Rd>,#<imm8>                    IT ブロック外部
MOV<c> <Rd>,#<imm8>                  IT ブロック内部
```

15 14 13	12 11	10 9 8	7 6 5 4 3 2 1 0
0 0 1	0 0	Rd	imm8

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

エンコード T2 ARMv6T2、ARMv7

```
MOV{S}<c>.W <Rd>,#<const>
```

15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	0	0 0 1 0	S	1 1 1 1	0	imm3	Rd	imm8

```
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

```
MOVW<c> <Rd>,#<imm16>
```

15 14 13 12 11	10	9	8	7	6	5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	1	0	0	1	0	0	imm4	0	imm3	Rd	imm8

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
if BadReg(d) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MOV{S}<c> <Rd>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	1 1 0 1	S	(0) (0) (0) (0)	Rd	imm12

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```

エンコード A2 ARMv6T2、ARMv7

```
MOVW<c> <Rd>,#<imm16>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0 1 1	0 0 0 0	imm4	Rd	imm12

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:imm12, 32);
if d == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`MOV{S}<c><q> <Rd>, #<const>`	All encodings permitted
`MOVW<c><q> <Rd>, #<const>`	Only encoding T3 or A2 permitted

where:

`S` If `S` is present, the instruction updates the flags. Otherwise, the flags are not updated.

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<const>` The immediate value to be placed in `<Rd>`. The range of values is 0-255 for encoding T1 and 0-65535 for encoding T3 or A2. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values for encoding T2 or A1.

When both 32-bit encodings are available for an instruction, encoding T2 or A1 is preferred to encoding T3 or A2 (if encoding T3 or A2 is required, use the `MOVW` syntax).

The pre-UAL syntax `MOV<c>S` is equivalent to `MOVS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = imm32;
    if d == 15 then          // Can only occur for encoding A1
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`MOV{S}<c><q> <Rd>, #<const>`	すべてのエンコードが許可される
`MOVW<c><q> <Rd>, #<const>`	エンコード T3 または A2 のみ許可される

各項目の説明については以下を参照して下さい。

`S` `S` を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<const>` `<Rd>` に書き込まれるイミディエート値。値の範囲は、エンコード T1 では 0 ～ 255、エンコード T3 または A2 では 0 ～ 65535 です。エンコード T2 または A1 での値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

両方の 32 ビットエンコードが使用可能な場合は、エンコード T3 または A2 よりもエンコード T2 または A1 の方が適しています（エンコード T3 または A2 が必要な場合は、`MOVW` 構文を使用して下さい）。

UAL 以前の構文の `MOV<c>S` は、`MOVS<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = imm32;
    if d == 15 then          // Can only occur for encoding A1
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.97 MOV (register)

Move (register) copies a value from a register to the destination register. It can optionally update the condition flags based on the value.

Encoding T1 ARMv6*, ARMv7 if <Rd> and <Rm> both from R0-R7
ARMv4T, ARMv5T*, ARMv6*, ARMv7 otherwise

```
MOV<c> <Rd>,<Rm>
```

If <Rd> is the PC, must be outside or last in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		

```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T2 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
MOVS <Rd>,<Rm>
```

Not permitted in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = TRUE;
if InITBlock() then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

```
MOV{S}<c>.W <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	0	0	0	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
if (d == 13 || BadReg(m)) && setflags then UNPREDICTABLE;
if (d == 13 && BadReg(m)) || d == 15 then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MOV{S}<c> <Rd>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				0	0	0	0	0	0	0	0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
```

Assembler syntax

```
MOV{S}<c><q>  <Rd>, <Rm>
```

where:

S If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.97 MOV（レジスタ）

MOV（移動（レジスタ））は、レジスタからデスティネーションレジスタに値をコピーします。オプションとして、その値に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6*、ARMv7（<Rd> と <Rm> の両方が R0 ～ R7 の場合）
ARMv4T、ARMv5T*、ARMv6*、ARMv7（上記以外の場合）

```
MOV<c> <Rd>,<Rm>
```

<Rd> が PC の場合は、IT ブロックの外部に配置するか、IT ブロック内部の最後の命令にする必要があります。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		

```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
MOVS <Rd>,<Rm>
```

IT ブロック内部では許可されない。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = TRUE;
if InITBlock() then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

```
MOV{S}<c>.W <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	0	0	0	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
if (d == 13 || BadReg(m)) && setflags then UNPREDICTABLE;
if (d == 13 && BadReg(m)) || d == 15 then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MOV{S}<c> <Rd>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				0	0	0	0	0	0	0	0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
```

アセンブラ構文

```
MOV{S}<c><q>  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

S S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register. This register can be the SP or PC. If this register is the PC and S is specified, see *SUBS PC, LR and related instructions* on page B6-25.

If <Rd> is the PC:

- the instruction causes a branch to the address moved to the PC
- in the Thumb and ThumbEE instruction sets:
  - — the instruction must either be outside an IT block or the last instruction of an IT block
  - — encoding T3 is not permitted.

In the Thumb and ThumbEE instruction sets, S must not be specified if <Rd> is the SP. If <Rd> is the SP and <Rm> is the SP or PC, encoding T3 is not permitted.

<Rm> The source register. This register can be the SP or PC. In the Thumb and ThumbEE instruction sets, S must not be specified if <Rm> is the SP or PC.

Note

The use of the following MOV (register) instructions is deprecated:

- ones in which <Rd> is the SP or PC and <Rm> is also the SP or PC
- ones in which S is specified and <Rd> is the SP, <Rm> is the SP, or <Rm> is the PC.

See also *Changing between Thumb state and ARM state* on page A4-2 about the use of the MOV PC,LR instruction.

The pre-UAL syntax MOV<c>S is equivalent to MOVS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[m];
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<c><q> 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ。SP または PC を指定できます。このレジスタが PC で、S を指定する場合は、*SUBS PC、LR、および関連する命令* P. B6-25 を参照して下さい。

<Rd> が PC の場合、次の動作が行われます。

- 命令によって、PC に移動されたアドレスへの分岐が発生します。
- Thumb および ThumbEE 命令セットでは、次の条件が適用されます。
  - – この命令は、IT ブロック外部に配置するか、IT ブロック内部の最後の命令にする必要があります。
  - – エンコード T3 は許可されません。

Thumb および ThumbEE 命令セットでは、<Rd> が SP である場合に S を指定することはできません。<Rd> が SP であり、<Rm> が SP または PC である場合は、エンコード T3 は許可されません。

<Rm> ソースレジスタ。SP または PC を指定できます。Thumb および ThumbEE 命令セットでは、<Rm> が SP または PC である場合に S を指定することはできません。

注

次の MOV（レジスタ）命令の使用は推奨されません。

- <Rd> が SP または PC であり、<Rm> も SP または PC であるような MOV 命令
- S が指定されており、かつ、<Rd> が SP であるか、<Rm> が SP であるか、または <Rm> が PC であるような MOV 命令

MOV PC,LR 命令の使用については、*Thumb 状態と ARM 状態との切り替え* P. A4-2 も参照して下さい。

UAL 以前の構文の MOV<c>S は、MOVS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[m];
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.98 MOV (shifted register)

Move (shifted register) is a pseudo-instruction for ASR, LSL, LSR, ROR, and RRX.

For details see the following sections:

- *ASR (immediate)* on page A8-40
- *ASR (register)* on page A8-42
- *LSL (immediate)* on page A8-178
- *LSL (register)* on page A8-180
- *LSR (immediate)* on page A8-182
- *LSR (register)* on page A8-184
- *ROR (immediate)* on page A8-278
- *ROR (register)* on page A8-280
- *RRX* on page A8-282.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.98 MOV（シフトしたレジスタ）

MOV（移動（シフトしたレジスタ））は、ASR、LSL、LSR、ROR、RRX の擬似命令です。

詳細については、次のセクションを参照して下さい。

- *ASR（イミディエート）* P. A8- 40
- *ASR（レジスタ）* P. A8- 42
- *LSL（イミディエート）* P. A8- 178
- *LSL（レジスタ）* P. A8- 180
- *LSR（イミディエート）* P. A8- 182
- *LSR（レジスタ）* P. A8- 184
- *ROR（イミディエート）* P. A8- 278
- *ROR（レジスタ）* P. A8- 280
- *RRX* P. A8- 282

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

Table A8-4 shows the equivalences between MOV (shifted register) and other instructions.

Table A8-4 MOV (shifted register) equivalences

MOV instruction	Canonical form
MOV{S} <Rd>,<Rm>,ASR #<n>	ASR{S} <Rd>,<Rm>,#<n>
MOV{S} <Rd>,<Rm>,LSL #<n>	LSL{S} <Rd>,<Rm>,#<n>
MOV{S} <Rd>,<Rm>,LSR #<n>	LSR{S} <Rd>,<Rm>,#<n>
MOV{S} <Rd>,<Rm>,ROR #<n>	ROR{S} <Rd>,<Rm>,#<n>
MOV{S} <Rd>,<Rm>,ASR <Rs>	ASR{S} <Rd>,<Rm>,<Rs>
MOV{S} <Rd>,<Rm>,LSL <Rs>	LSL{S} <Rd>,<Rm>,<Rs>
MOV{S} <Rd>,<Rm>,LSR <Rs>	LSR{S} <Rd>,<Rm>,<Rs>
MOV{S} <Rd>,<Rm>,ROR <Rs>	ROR{S} <Rd>,<Rm>,<Rs>
MOV{S} <Rd>,<Rm>,RRX	RRX{S} <Rd>,<Rm>

Disassembly produces the canonical form of the instruction.

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

MOV（シフトしたレジスタ）と他の命令の対応を、表 A8- 4 に示します。

表 A8- 4 MOV（シフトしたレジスタ）と他の命令の対応

MOV 命令	標準形式
MOV{S} <Rd>,<Rm>,ASR #<n>	ASR{S} <Rd>,<Rm>,#<n>
MOV{S} <Rd>,<Rm>,LSL #<n>	LSL{S} <Rd>,<Rm>,#<n>
MOV{S} <Rd>,<Rm>,LSR #<n>	LSR{S} <Rd>,<Rm>,#<n>
MOV{S} <Rd>,<Rm>,ROR #<n>	ROR{S} <Rd>,<Rm>,#<n>
MOV{S} <Rd>,<Rm>,ASR <Rs>	ASR{S} <Rd>,<Rm>,<Rs>
MOV{S} <Rd>,<Rm>,LSL <Rs>	LSL{S} <Rd>,<Rm>,<Rs>
MOV{S} <Rd>,<Rm>,LSR <Rs>	LSR{S} <Rd>,<Rm>,<Rs>
MOV{S} <Rd>,<Rm>,ROR <Rs>	ROR{S} <Rd>,<Rm>,<Rs>
MOV{S} <Rd>,<Rm>,RRX	RRX{S} <Rd>,<Rm>

命令の標準形式は、逆アセンブリによって生成されます。

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.99 MOVT

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

Encoding T1 ARMv6T2, ARMv7

```
MOVT<c> <Rd>,#<imm16>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	1	0	0	imm4				0	imm3			Rd				imm8							

```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if BadReg(d) then UNPREDICTABLE;
```

Encoding A1 ARMv6T2, ARMv7

```
MOVT<c> <Rd>,#<imm16>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	1	0	0	imm4				Rd				imm12											

```
d = UInt(Rd);  imm16 = imm4:imm12;
if d == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.99 MOVT

MOVT（上位ビットへの移動）は、イミディエート値をデスティネーションレジスタの上位ハーフワードに書き込みます。下位ハーフワードの内容には影響を与えません。

エンコード T1 ARMv6T2、ARMv7

```
MOVT<c> <Rd>,#<imm16>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	1	0	0	imm4				0	imm3			Rd				imm8							

```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if BadReg(d) then UNPREDICTABLE;
```

エンコード A1 ARMv6T2、ARMv7

```
MOVT<c> <Rd>,#<imm16>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	1	0	0	imm4				Rd				imm12											

```
d = UInt(Rd);  imm16 = imm4:imm12;
if d == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MOVT<c><q>  <Rd>, #<imm16>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<imm16> The immediate value to be written to <Rd>. It must be in the range 0-65535.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<31:16> = imm16;
    // R[d]<15:0> unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MOVT<c><q>  <Rd>, #<imm16>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<imm16> <Rd> に書き込まれるイミディエート値。許容される値の範囲は 0 ～ 65535 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<31:16> = imm16;
    // R[d]<15:0> unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.100 MRC, MRC2

Move to ARM core register from Coprocessor causes a coprocessor to transfer a value to an ARM core register or to the condition flags. If no coprocessor can execute the instruction, an Undefined Instruction exception is generated.

This is a generic coprocessor instruction. Some of the fields have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer. These fields are the opc1, opc2, CRn, and CRm fields.

For more information about the coprocessors see *Coprocessor support* on page A2-68.

Encoding T1 / A1 ARMv6T2, ARMv7 for encoding T1
ARMv4*, ARMv5T*, ARMv6*, ARMv7 for encoding A1

```
MRC<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}
```

15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 0	1 1 1 0	opc1	1	CRn	Rt	coproc	opc2	1	CRm

31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
cond	1 1 1 0	opc1	1	CRn	Rt	coproc	opc2	1	CRm

```
if coproc == '101x' then SEE "Advanced SIMD and VFP";
t = UInt(Rt);  cp = UInt(coproc);
if t == 13 && (CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

Encoding T2 / A2 ARMv6T2, ARMv7 for encoding T2
ARMv5T*, ARMv6*, ARMv7 for encodingA2

```
MRC2<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}
```

15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1	1 1 1 0	opc1	1	CRn	Rt	coproc	opc2	1	CRm

31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1	1 1 1 0	opc1	1	CRn	Rt	coproc	opc2	1	CRm

```
t = UInt(Rt);  cp = UInt(coproc);
if t == 13 && (CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

Advanced SIMD and VFP See *8, 16, and 32-bit transfer between ARM core and extension registers* on page A7-31

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.100 MRC、MRC2

MRC、MRC2（コプロセッサから ARM コアレジスタへの移動）は、コプロセッサに対して、ARM コアレジスタまたは条件フラグへ値を転送するように指示します。どのコプロセッサも命令を実行できない場合は、未定義命令例外が発生します。

これは、汎用のコプロセッサ命令です。一部のフィールドの機能はアーキテクチャで定義されていないため、コプロセッサ命令セットの設計者が自由に使用することができます。これらのフィールドは、opc1、opc2、CRn、CRm です。

コプロセッサの詳細については、*コプロセッサのサポート* P. A2-68 を参照して下さい。

エンコード T1/A1 ARMv6T2、ARMv7（エンコード T1 の場合）
ARMv4*、ARMv5T*、ARMv6*、ARMv7（エンコード A1 の場合）

```
MRC<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}
```

15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 0	1 1 1 0	opc1	1	CRn	Rt	coproc	opc2	1	CRm

31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
cond	1 1 1 0	opc1	1	CRn	Rt	coproc	opc2	1	CRm

```
if coproc == '101x' then SEE "Advanced SIMD and VFP";
t = UInt(Rt);  cp = UInt(coproc);
if t == 13 && (CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

エンコード T2/A2 ARMv6T2、ARMv7（エンコード T2 の場合）
ARMv5T*、ARMv6*、ARMv7（エンコード A2 の場合）

```
MRC2<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}
```

15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1	1 1 1 0	opc1	1	CRn	Rt	coproc	opc2	1	CRm

31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1	1 1 1 0	opc1	1	CRn	Rt	coproc	opc2	1	CRm

```
t = UInt(Rt);  cp = UInt(coproc);
if t == 13 && (CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

アドバンスト SIMD および VFP *ARM コアレジスタと拡張機能レジスタとの 8、16、32 ビット転送* P. A7-31 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MRC{2}<c><q>  <coproc>, #<opc1>, <Rt>, <CRn>, <CRm>{, #<opc2>}
```

where:

2 — If specified, selects encoding T2 / A2. If omitted, selects encoding T1 / A1.

<c><q> — See *Standard assembler syntax fields* on page A8-7. An ARM MRC2 instruction must be unconditional.

<coproc> — The name of the coprocessor. The standard generic coprocessor names are p0, p1, …, p15.

<opc1> — Is a coprocessor-specific opcode in the range 0 to 7.

<Rt> — Is the destination ARM core register. This register can be R0-R14 or APSR_nzcv. The last form writes bits [31:28] of the transferred value to the N, Z, C and V condition flags and is specified by setting the Rt field of the encoding to 0b1111. In pre-UAL assembler syntax, PC was written instead of APSR_nzcv to select this form.

<CRn> — Is the coprocessor register that contains the first operand.

<CRm> — Is an additional source or destination coprocessor register.

<opc2> — Is a coprocessor-specific opcode in the range 0 to 7. If omitted, <opc2> is assumed to be 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        value = Coproc_GetOneWord(cp, ThisInstr());
        if t != 15 then
            R[t] = value;
        else
            APSR.N = value<31>;
            APSR.Z = value<30>;
            APSR.C = value<29>;
            APSR.V = value<28>;
            // value<27:0> are not used.
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MRC{2}<c><q>  <coproc>, #<opc1>, <Rt>, <CRn>, <CRm>{, #<opc2>}
```

各項目の説明については以下を参照して下さい。

2 — 指定した場合、エンコード T2/A2 が選択されます。省略した場合、エンコード T1/A1 が選択されます。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM MRC2 は常に無条件命令です。

<coproc> — コプロセッサの名前。標準の汎用コプロセッサ名は、p0、p1、...、p15 です。

<opc1> — コプロセッサ固有のオペコードで、範囲は 0 ～ 7 です。

<Rt> — デスティネーションの ARM コアレジスタ。R0 ～ R14 または APSR_nzcv を指定できます。最後の形式は、転送される値のビット [31:28] を、N、Z、C、V の条件フラグに書き込みます。これは、エンコードの Rt フィールドを 0b1111 に設定することによって指定されます。UAL 以前のアセンブラ構文では、この形式を選択するために APSR_nzcv の代わりに PC が書き込まれます。

<CRn> — 最初のオペランドを含むコプロセッサレジスタ

<CRm> — 追加のソースまたはデスティネーション コプロセッサレジスタ

<opc2> — コプロセッサ固有のオペコードで、範囲は 0 ～ 7 です。省略した場合、<opc2> が 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        value = Coproc_GetOneWord(cp, ThisInstr());
        if t != 15 then
            R[t] = value;
        else
            APSR.N = value<31>;
            APSR.Z = value<30>;
            APSR.C = value<29>;
            APSR.V = value<28>;
            // value<27:0> are not used.
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.101 MRRC, MRRC2

Move to two ARM core registers from Coprocessor causes a coprocessor to transfer values to two ARM core registers. If no coprocessor can execute the instruction, an Undefined Instruction exception is generated.

This is a generic coprocessor instruction. The opc1 and CRm fields have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer.

For more information about the coprocessors see *Coprocessor support* on page A2-68.

Encoding T1 / A1 ARMv6T2, ARMv7 for encoding T1
ARMv5TE*, ARMv6*, ARMv7 for encoding A1

```
MRRC<c> <coproc>,<opc>,<Rt>,<Rt2>,<CRm>
```

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0	1 1 0	0	0	1	0	1	Rt2	Rt	coproc	opc1	CRm

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	1 1 0	0	0	1	0	1	Rt2	Rt	coproc	opc1	CRm

```
if coproc == '101x' then SEE "Advanced SIMD and VFP";
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
if (t == 13 || t2 == 13) && (CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

Encoding T2 / A2 ARMv6T2, ARMv7 for encoding T2
ARMv6*, ARMv7 for encoding A2

```
MRRC2<c> <coproc>,<opc>,<Rt>,<Rt2>,<CRm>
```

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	1 1 0	0	0	1	0	1	Rt2	Rt	coproc	opc1	CRm

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	1 1 0	0	0	1	0	1	Rt2	Rt	coproc	opc1	CRm

```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
if (t == 13 || t2 == 13) && (CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

Advanced SIMD and VFP See *64-bit transfers between ARM core and extension registers* on page A7-32

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.101 MRRC、MRRC2

MRRC、MRRC2（コプロセッサから 2 つの ARM コアレジスタへの移動）は、コプロセッサに対して、2 つの ARM コアレジスタへ値を転送するように指示します。どのコプロセッサも命令を実行できない場合は、未定義命令例外が発生します。

これは、汎用のコプロセッサ命令です。opc1 フィールドと CRm フィールドの機能はアーキテクチャで定義されていないため、コプロセッサ命令セットの設計者が自由に使用することができます。

コプロセッサの詳細については、*コプロセッサのサポート* P. A2-68 を参照して下さい。

エンコード T1/A1 ARMv6T2、ARMv7（エンコード T1 の場合）
ARMv5TE*、ARMv6*、ARMv7（エンコード A1 の場合）

```
MRRC<c> <coproc>,<opc>,<Rt>,<Rt2>,<CRm>
```

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0	1 1 0	0	0	1	0	1	Rt2	Rt	coproc	opc1	CRm

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	1 1 0	0	0	1	0	1	Rt2	Rt	coproc	opc1	CRm

```
if coproc == '101x' then SEE "Advanced SIMD and VFP";
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
if (t == 13 || t2 == 13) && (CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

エンコード T2/A2 ARMv6T2、ARMv7（エンコード T2 の場合）
ARMv6*、ARMv7（エンコード A2 の場合）

```
MRRC2<c> <coproc>,<opc>,<Rt>,<Rt2>,<CRm>
```

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	1 1 0	0	0	1	0	1	Rt2	Rt	coproc	opc1	CRm

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1	1 1 0	0	0	1	0	1	Rt2	Rt	coproc	opc1	CRm

```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
if (t == 13 || t2 == 13) && (CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

アドバンスト SIMD および VFP *ARM コアレジスタと拡張機能レジスタとの 64 ビット転送* P. A7-32 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MRRC{2}<c><q>  <coproc>, #<opc1>, <Rt>, <Rt2>, <CRm>
```

where:

2	If specified, selects encoding T2 / A2. If omitted, selects encoding T1 / A1.
<c><q>	See *Standard assembler syntax fields* on page A8-7. An ARM MRRC2 instruction must be unconditional.
<coproc>	The name of the coprocessor. The standard generic coprocessor names are p0, p1, …, p15.
<opc1>	Is a coprocessor-specific opcode in the range 0 to 15.
<Rt>	Is the first destination ARM core register.
<Rt2>	Is the second destination ARM core register.
<CRm>	Is the coprocessor register that supplies the data to be transferred.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        (R[t], R[t2]) = Coproc_GetTwoWords(cp, ThisInstr());
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MRRC{2}<c><q>  <coproc>, #<opc1>, <Rt>, <Rt2>, <CRm>
```

各項目の説明については以下を参照して下さい。

2	指定した場合、エンコード T2/A2 が選択されます。省略した場合、エンコード T1/A1 が選択されます。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM MRRC2 は常に無条件命令です。
<coproc>	コプロセッサの名前。標準の汎用コプロセッサ名は、p0、p1、...、p15 です。
<opc1>	コプロセッサ固有のオペコードで、範囲は 0 ～ 15 です。
<Rt>	最初のデスティネーション ARM コアレジスタ
<Rt2>	2 番目のデスティネーション ARM コアレジスタ
<CRm>	転送されるデータを供給するコプロセッサレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        (R[t], R[t2]) = Coproc_GetTwoWords(cp, ThisInstr());
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.102 MRS

Move to Register from Special Register moves the value from the APSR into a general-purpose register.

For details of system level use of this instruction, see *MRS* on page B6-10.

Encoding T1 ARMv6T2, ARMv7

```
MRS<c> <Rd>,<spec_reg>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	Rd				(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
d = UInt(Rd);
if BadReg(d) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MRS<c> <Rd>,<spec_reg>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	0	0	(1)	(1)	(1)	(1)	Rd				(0)	(0)	(0)	(0)	0	0	0	0	(0)	(0)	(0)	(0)

```
d = UInt(Rd);
if d == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.102 MRS

MRS（特殊レジスタからレジスタへの移動）は、APSR の値を汎用レジスタに移動します。

この命令をシステムレベルで使用する方法の詳細については、*MRS* P. B6-10 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
MRS<c> <Rd>,<spec_reg>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	Rd				(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
d = UInt(Rd);
if BadReg(d) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MRS<c> <Rd>,<spec_reg>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	0	0	(1)	(1)	(1)	(1)	Rd				(0)	(0)	(0)	(0)	0	0	0	0	(0)	(0)	(0)	(0)

```
d = UInt(Rd);
if d == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MRS<c><q>  <Rd>, <spec_reg>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<spec_reg>` Is one of:
- `APSR`
- `CPSR`.

ARM recommends the `APSR` form in application level code. For more information, see *The Application Program Status Register (APSR)* on page A2-14.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d] = APSR;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MRS<c><q>  <Rd>, <spec_reg>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<spec_reg>` 次のいずれかを指定します。
- `APSR`
- `CPSR`

アプリケーションレベルのコードでは `APSR` 形式を使用することを推奨します。詳細については、*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2- 14 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d] = APSR;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.103 MSR (immediate)

Move immediate value to Special Register moves selected bits of an immediate value to the corresponding bits in the APSR.

For details of system level use of this instruction, see *MSR (immediate)* on page B6-12.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MSR<c> <spec_reg>,#<const>
```

31 30 29 28	27 26 25 24 23	22	21 20	19 18	17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0 1 1 0	0	1 0	mask	0 0	(1) (1) (1) (1)	imm12

```
if mask == '00' then SEE "Related encodings";
imm32 = ARMExpandImm(imm12);  write_nzcvq = (mask<1> == '1');  write_g = (mask<0> == '1');
if n == 15 then UNPREDICTABLE;
```

Assembler syntax

```
MSR<c><q>  <spec_reg>, #<imm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<spec_reg>` Is one of:

- `APSR_<bits>`
- `CPSR_<fields>`.

ARM recommends the `APSR` forms in application level code. For more information, see *The Application Program Status Register (APSR)* on page A2-14.

`<imm>` Is the immediate value to be transferred to `<spec_reg>`. See *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

`<bits>` Is one of `nzcvq`, `g`, or `nzcvqg`.

In the A and R profiles:

- `APSR_nzcvq` is the same as `CPSR_f`
- `APSR_g` is the same as `CPSR_s`
- `APSR_nzcvqg` is the same as `CPSR_fs`.

`<fields>` Is a sequence of one or more of the following: `s`, `f`.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.103 MSR（イミディエート）

MSR（イミディエート値の特殊レジスタへの移動（イミディエート））は、イミディエート値の選択されたビットを、APSR の対応するビットに移動します。

この命令をシステムレベルで使用する方法の詳細については、*MSR（イミディエート）* P. B6-12 を参照して下さい。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MSR<c> <spec_reg>,#<const>
```

31 30 29 28	27 26 25 24 23	22	21 20	19 18	17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0 1 1 0	0	1 0	mask	0 0	(1) (1) (1) (1)	imm12

```
if mask == '00' then SEE "Related encodings";
imm32 = ARMExpandImm(imm12);  write_nzcvq = (mask<1> == '1');  write_g = (mask<0> == '1');
if n == 15 then UNPREDICTABLE;
```

Assembler syntax

```
MSR<c><q>  <spec_reg>, #<imm>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<spec_reg>` 次のいずれかを指定します。

- `APSR_<bits>`
- `CPSR_<fields>`

アプリケーションレベルのコードでは `APSR` 形式を使用することを推奨します。詳細については、*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。

`<imm>` `<spec_reg>` に転送されるイミディエート値。値の範囲については、*ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

`<bits>` `nzcvq`、`g`、`nzcvqg` のいずれかを指定します。

A プロファイルと R プロファイルでは、次の条件が適用されます。

- `APSR_nzcvq` は `CPSR_f` と同じです。
- `APSR_g` は `CPSR_s` と同じです。
- `APSR_nzcvqg` は `CPSR_fs` と同じです。

`<fields>` `s` または `f` の 1 つまたは複数のシーケンス

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_nzcvq then
        APSR.N = imm32<31>;
        APSR.Z = imm32<30>;
        APSR.C = imm32<29>;
        APSR.V = imm32<28>;
        APSR.Q = imm32<27>;
    if write_g then
        APSR.GE = imm32<19:16>;
```

Exceptions

None.

Usage

For details of the APSR see *The Application Program Status Register (APSR)* on page A2-14. Because of the Do-Not-Modify nature of its reserved bits, the immediate form of MSR is normally only useful at the Application level for writing to APSR_nzcvq (CPSR_f).

For the A and R profiles, *MSR (immediate)* on page B6-12 describes additional functionality that is available using the reserved bits. This includes some deprecated functionality that is available in unprivileged and privileged modes and therefore can be used at the Application level.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_nzcvq then
        APSR.N = imm32<31>;
        APSR.Z = imm32<30>;
        APSR.C = imm32<29>;
        APSR.V = imm32<28>;
        APSR.Q = imm32<27>;
    if write_g then
        APSR.GE = imm32<19:16>;
```

例外

なし

使用法

APSR の詳細については、*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。この予約ビットには「変更不可」という特性があるため、イミディエート形式の MSR は通常、アプリケーションレベルでの APSR_nzcvq (CPSR_f) への書き込みにのみ使用できます。

A プロファイルと R プロファイルについては、予約ビットによって使用できる追加機能が *MSR (イミディエート)* P. B6-12 に示されています。この中には、非特権モードと特権モードで使用できる、したがってアプリケーションレベルで使用できる、推奨されない機能が含まれています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.104 MSR (register)

Move to Special Register from ARM core register moves selected bits of a general-purpose register to the APSR.

For details of system level use of this instruction, see *MSR (register)* on page B6-14.

Encoding T1 ARMv6T2, ARMv7

```
MSR<c> <spec_reg>,<Rn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	Rn				1	0	(0)	0	mask		0	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
n = UInt(Rn);  write_nzcvq = (mask<1> == '1');  write_g = (mask<0> == '1');
if mask == '00' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MSR<c> <spec_reg>,<Rn>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	1	0	mask		0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	Rn			

```
n = UInt(Rn);  write_nzcvq = (mask<1> == '1');  write_g = (mask<0> == '1');
if mask == '00' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

Assembler syntax

```
MSR<c><q>  <spec_reg>, <Rn>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<spec_reg>` Is one of:
- `APSR_<bits>`
- `CPSR_<fields>`.

ARM recommends the `APSR` forms in application level code. For more information, see *The Application Program Status Register (APSR)* on page A2-14.

`<Rn>` Is the general-purpose register to be transferred to `<spec_reg>`.

`<bits>` Is one of `nzcvq`, `g`, or `nzcvqg`.

In the A and R profiles:
- `APSR_nzcvq` is the same as `CPSR_f`
- `APSR_g` is the same as `CPSR_s`
- `APSR_nzcvqg` is the same as `CPSR_fs`.

`<fields>` Is a sequence of one or more of the following: `s`, `f`.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.104 MSR（レジスタ）

MSR（ARM コアレジスタから特殊レジスタへの移動（レジスタ））は、汎用レジスタの選択されたビットを APSR へ移動します。

この命令をシステムレベルで使用する方法の詳細については、*MSR（レジスタ）* P. B6-14 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
MSR<c> <spec_reg>,<Rn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	Rn				1	0	(0)	0	mask		0	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
n = UInt(Rn);  write_nzcvq = (mask<1> == '1');  write_g = (mask<0> == '1');
if mask == '00' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MSR<c> <spec_reg>,<Rn>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	1	0	mask		0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	Rn			

```
n = UInt(Rn);  write_nzcvq = (mask<1> == '1');  write_g = (mask<0> == '1');
if mask == '00' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

アセンブラ構文

```
MSR<c><q>  <spec_reg>, <Rn>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<spec_reg>` 次のいずれかを指定します。
- `APSR_<bits>`
- `CPSR_<fields>`

アプリケーションレベルのコードでは `APSR` 形式を使用することを推奨します。詳細については、*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。

`<Rn>` `<spec_reg>` に転送される汎用レジスタ

`<bits>` `nzcvq`、`g`、`nzcvqg` のいずれかを指定します。

A プロファイルと R プロファイルでは、次の条件が適用されます。
- `APSR_nzcvq` は `CPSR_f` と同じです。
- `APSR_g` は `CPSR_s` と同じです。
- `APSR_nzcvqg` は `CPSR_fs` と同じです。

`<fields>` `s` または `f` の 1 つまたは複数のシーケンス

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_nzcvq then
        APSR.N = R[n]<31>;
        APSR.Z = R[n]<30>;
        APSR.C = R[n]<29>;
        APSR.V = R[n]<28>;
        APSR.Q = R[n]<27>;
    if write_g then
        APSR.GE = R[n]<19:16>;
```

Exceptions

None.

Usage

For details of the APSR see *The Application Program Status Register (APSR)* on page A2-14. Because of the Do-Not-Modify nature of its reserved bits, a read / modify / write sequence is normally needed when the `MSR` instruction is being used at Application level and its destination is not `APSR_nzcvq` (`CPSR_f`).

For the A and R profiles, *MSR (register)* on page B6-14 describes additional functionality that is available using the reserved bits. This includes some deprecated functionality that is available in unprivileged and privileged modes and therefore can be used at the Application level.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_nzcvq then
        APSR.N = R[n]<31>;
        APSR.Z = R[n]<30>;
        APSR.C = R[n]<29>;
        APSR.V = R[n]<28>;
        APSR.Q = R[n]<27>;
    if write_g then
        APSR.GE = R[n]<19:16>;
```

例外

なし

使用法

APSR の詳細については、*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。この予約ビットには「変更不可」という特性があるため、通常は、`MSR` 命令がアプリケーションレベルで使用され、そのデスティネーションが `APSR_nzcvq` (`CPSR_f`) ではないときに、読み出し－変更－書き込みシーケンスが必要です。

A プロファイルと R プロファイルについては、予約ビットによって使用できる追加機能が *MSR (レジスタ)* P. B6-14 に示されています。この中には、非特権モードと特権モードで使用できる、したがってアプリケーションレベルで使用できる、推奨されない機能が含まれています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.105 MUL

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

Optionally, it can update the condition flags based on the result. In the Thumb instruction set, this option is limited to only a few forms of the instruction. Use of this option adversely affects performance on many processor implementations.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
MULS <Rdm>,<Rn>,<Rdm>                    Outside IT block.
MUL<c> <Rdm>,<Rn>,<Rdm>                  Inside IT block.
```

15 14 13 12 11 10	9 8 7 6	5 4 3	2 1 0
0 1 0 0 0 0	1 1 0 1	Rn	Rdm

```
d = UInt(Rdm);  n = UInt(Rn);  m = UInt(Rdm);  setflags = !InITBlock();
if ArchVersion() < 6 && d == n then UNPREDICTABLE;
```

Encoding T2 ARMv6T2, ARMv7

```
MUL<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11 10	9 8 7 6	5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 0 0	Rn	1 1 1 1	Rd	0 0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MUL{S}<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 0 0 0	S	Rd	(0) (0) (0) (0)	Rm	1 0 0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && d == n then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.105 MUL

MUL（乗算）は、2 つのレジスタの値を乗算します。結果の下位 32 ビットがデスティネーションレジスタに書き込まれます。この 32 ビットは、ソースレジスタの値が符号付きの値または符号なしの値のどちらであると見なされるかに依存しません。

オプションとして、結果に基づいて条件フラグを更新することができます。Thumb 命令セットでは、このオプションは少数の形式の命令に限定されます。このオプションを使用すると、多くのプロセッサ実装で性能が低下します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
MULS <Rdm>,<Rn>,<Rdm>                    IT ブロック外部
MUL<c> <Rdm>,<Rn>,<Rdm>                  IT ブロック内部
```

15 14 13 12 11 10	9 8 7 6	5 4 3	2 1 0
0 1 0 0 0 0	1 1 0 1	Rn	Rdm

```
d = UInt(Rdm);  n = UInt(Rn);  m = UInt(Rdm);  setflags = !InITBlock();
if ArchVersion() < 6 && d == n then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

```
MUL<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11 10	9 8 7 6	5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 0 0	Rn	1 1 1 1	Rd	0 0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MUL{S}<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 0 0 0	S	Rd	(0) (0) (0) (0)	Rm	1 0 0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && d == n then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MUL{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

In the Thumb instruction set, S can be specified only if both <Rn> and <Rm> are R0-R7 and the instruction is outside an IT block.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
        // else APSR.C unchanged
        // APSR.V always unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MUL{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

Thumb 命令セットでは、<Rn> と <Rm> の両方が R0 ～ R7 であり、命令が IT ブロックの外部にある場合のみ、S を指定できます。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — 2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
        // else APSR.C unchanged
        // APSR.V always unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.106 MVN (immediate)

Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register. It can optionally update the condition flags based on the value.

Encoding T1 ARMv6T2, ARMv7

```
MVN{S}<c> <Rd>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	1	S	1	1	1	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MVN{S}<c> <Rd>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	1	1	1	S	(0)	(0)	(0)	(0)	Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.106 MVN（イミディエート）

MVN（ビット単位否定（イミディエート））は、イミディエート値のビット単位の否定をデスティネーションレジスタに書き込みます。オプションとして、その値に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

```
MVN{S}<c> <Rd>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	1	S	1	1	1	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MVN{S}<c> <Rd>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	1	1	1	S	(0)	(0)	(0)	(0)	Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MVN{S}<c><q>  <Rd>, #<const>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<const>	The immediate value to be bitwise inverted. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

The pre-UAL syntax MVN<c>S is equivalent to MVNS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = NOT(imm32);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MVN{S}<c><q>  <Rd>, #<const>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<const>	ビット単位で反転されるイミディエート値。値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

UAL 以前の構文の MVN<c>S は、MVNS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = NOT(imm32);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.107 MVN (register)

Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
MVNS <Rd>,<Rm>                    Outside IT block.
MVN<c> <Rd>,<Rm>                  Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
MVN{S}<c>.W <Rd>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	1	S	1	1	1	1	(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MVN{S}<c> <Rd>,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	1	1	S	(0)	(0)	(0)	(0)	Rd				imm5				type			0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.107 MVN（レジスタ）

MVN（ビット単位否定（レジスタ））は、レジスタの値のビット単位の否定をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
MVNS <Rd>,<Rm>                    IT ブロック外部
MVN<c> <Rd>,<Rm>                  IT ブロック内部
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
MVN{S}<c>.W <Rd>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	1	S	1	1	1	1	(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MVN{S}<c> <Rd>,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	1	1	S	(0)	(0)	(0)	(0)	Rd				imm5				type			0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MVN{S}<c><q>  <Rd>, <Rm> {, <shift>}
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rm>	The register that is optionally shifted and used as the source register.
<shift>	The shift to apply to the value read from <Rm>. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

The pre-UAL syntax MVN<c>S is equivalent to MVNS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = NOT(shifted);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MVN{S}<c><q>  <Rd>, <Rm> {, <shift>}
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rm>	オプションとしてシフトされ、ソースレジスタとして使用されるレジスタ
<shift>	<Rm> から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

UAL 以前の構文の MVN<c>S は、MVNS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = NOT(shifted);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.108 MVN (register-shifted register)

Bitwise NOT (register-shifted register) writes the bitwise inverse of a register-shifted register value to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MVN{S}<c> <Rd>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 1 1 1	S	(0) (0) (0) (0)	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.108 MVN（レジスタによりシフトしたレジスタ）

MVN（ビット単位否定（レジスタによりシフトしたレジスタ））は、レジスタによりシフトしたレジスタの値のビット単位の否定を、デスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MVN{S}<c> <Rd>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 1 1 1	S	(0) (0) (0) (0)	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MVN{S}<c><q>  <Rd>, <Rm>, <type>  <Rs>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rm> — The register that is shifted and used as the operand.

<type> — The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR — Arithmetic shift right, encoded as type = 0b10
- LSL — Logical shift left, encoded as type = 0b00
- LSR — Logical shift right, encoded as type = 0b01
- ROR — Rotate right, encoded as type = 0b11.

<Rs> — The register whose bottom byte contains the amount to shift by.

The pre-UAL syntax MVN<c>S is equivalent to MVNS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MVN{S}<c><q>  <Rd>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rm> — シフトされ、オペランドとして使用されるレジスタ

<type> — <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR — 算術右シフト。type = 0b10 としてエンコードされます。
- LSL — 論理左シフト。type = 0b00 としてエンコードされます。
- LSR — 論理右シフト。type = 0b01 としてエンコードされます。
- ROR — 右ローテート。type = 0b11 としてエンコードされます。

<Rs> — 最下位バイトにシフトのビット数を含むレジスタ

UAL 以前の構文の MVN<c>S は、MVNS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.109 NEG

Negate is a pre-UAL synonym for RSB (immediate) with an immediate value of 0. For details see *RSB (immediate)* on page A8-284.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.109 NEG

NEG（否定）は、UAL 以前のイミディエート値が 0 の RSB（イミディエート）と同義語です。
詳細については、*RSB（イミディエート）*P. A8- 284 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
NEG<c><q>  <Rd>, <Rm>
```

This is equivalent to:

```
RSBS<c><q>  <Rd>, <Rm>, #0
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
NEG<c><q>  <Rd>, <Rm>
```

これは、次の命令と等価です。

```
RSBS<c><q>  <Rd>, <Rm>, #0
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.110 NOP

No Operation does nothing. This instruction can be used for code alignment purposes.

See *Pre-UAL pseudo-instruction NOP* on page AppxC-3 for details of NOP before the introduction of UAL and the ARMv6K and ARMv6T2 architecture variants.

Note

The timing effects of including a NOP instruction in code are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. NOP instructions are therefore not suitable for timing loops.

Encoding T1 ARMv6T2, ARMv7

NOP<c>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0

```
// No additional decoding required
```

Encoding T2 ARMv6T2, ARMv7

NOP<c>.W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	0

```
// No additional decoding required
```

Encoding A1 ARMv6K, ARMv6T2, ARMv7

NOP<c>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	0	0

```
// No additional decoding required
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.110 NOP

NOP（無操作）は何の動作も行いません。この命令は、コードのアライメントのために使用することができます。

UAL と ARMv6K/ARMv6T2 アーキテクチャバリアントが導入される前の NOP の詳細については、*UAL 以前の擬似命令 NOP* P. 付録 C-3 を参照して下さい。

注

NOP 命令をコードに含めた場合のタイミング効果は保証されません。実行時間が長くなったり、そのままであったり、短くなったりすることもあります。したがって、NOP 命令はタイミングループには適していません。

エンコード T1 ARMv6T2、ARMv7

NOP<c>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0

```
// No additional decoding required
```

エンコード T2 ARMv6T2、ARMv7

NOP<c>.W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	0

```
// No additional decoding required
```

エンコード A1 ARMv6K、ARMv6T2、ARMv7

NOP<c>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	0	0

```
// No additional decoding required
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
NOP<c><q>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
NOP<c><q>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.111 ORN (immediate)

Bitwise OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv6T2, ARMv7

```
ORN{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	1	S	Rn				0	imm3			Rd				imm8							

```
if Rn == '1111' then SEE MVN (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || n == 13 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.111 ORN（イミディエート）

ORN（ビット単位の否定論理和（イミディエート））は、レジスタの値と、イミディエート値の補数とのビット単位の（非排他的）論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

```
ORN{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	1	S	Rn				0	imm3			Rd				imm8							

```
if Rn == '1111' then SEE MVN (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || n == 13 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ORN{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The register that contains the operand.
<const>	The immediate value to be bitwise inverted and ORed with the value obtained from <Rn>. See *Modified immediate constants in Thumb instructions* on page A6-17 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR NOT(imm32);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ORN{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	オペランドを含むレジスタ
<const>	ビット単位で反転され、<Rn> から取得された値との論理和が実行されるイミディエート値。値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6- 17 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR NOT(imm32);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.112 ORN (register)

Bitwise OR NOT (register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv6T2, ARMv7

```
ORN{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12 11	10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 0 1 1	S	Rn	(0)	imm3	Rd	imm2	type	Rm

```
if Rn == '1111' then SEE MVN (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.112 ORN（レジスタ）

ORN（ビット単位の否定論理和（レジスタ））は、レジスタの値と、オプションとしてシフトしたレジスタの値の補数とでビット単位（非排他的）論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

```
ORN{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12 11	10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 0 1 1	S	Rn	(0)	imm3	Rd	imm2	type	Rm

```
if Rn == '1111' then SEE MVN (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ORN{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The register that is optionally shifted and used as the second operand.
<shift>	The shift to apply to the value read from <Rm>. If omitted, no shift is applied. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ORN{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ
<shift>	<Rm> から読み出された値に適用されるシフト。省略した場合、シフトは適用されません。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.113 ORR (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv6T2, ARMv7

```
ORR{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	0	S	Rn				0	imm3			Rd				imm8							

```
if Rn == '1111' then SEE MOV (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || n == 13 then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ORR{S}<c> <Rd>,<Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	1	0	0	S	Rn				Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.113 ORR（イミディエート）

ORR（ビット単位の論理和（イミディエート））は、レジスタの値とイミディエート値とのビット単位の（非排他的）論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

```
ORR{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	0	S	Rn				0	imm3			Rd				imm8							

```
if Rn == '1111' then SEE MOV (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || n == 13 then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ORR{S}<c> <Rd>,<Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	1	0	0	S	Rn				Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ORR{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The register that contains the operand.
<const>	The immediate value to be bitwise ORed with the value obtained from <Rn>. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

The pre-UAL syntax ORR<c>S is equivalent to ORRS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR imm32;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ORR{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	オペランドを含むレジスタ
<const>	<Rn> から取得された値とのビット単位の論理和が実行されるイミディエート値。値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

UAL 以前の構文の ORR<c>S は、ORRS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR imm32;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.114 ORR (register)

Bitwise OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ORRS    <Rdn>,<Rm>                    Outside IT block.
ORR<c> <Rdn>,<Rm>                     Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
ORR{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
if Rn == '1111' then SEE MOV (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ORR{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	0	S	Rn				Rd				imm5				type			0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.114 ORR（レジスタ）

ORR（ビット単位の論理和（レジスタ））は、レジスタの値と、オプションとしてシフトされたレジスタの値とのビット単位の（非排他的）論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
ORRS    <Rdn>,<Rm>                    IT ブロック外部
ORR<c> <Rdn>,<Rm>                     IT ブロック内部
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
ORR{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
if Rn == '1111' then SEE MOV (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ORR{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	0	S	Rn				Rd				imm5				type			0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ORR{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The register that is optionally shifted and used as the second operand.
<shift>	The shift to apply to the value read from <Rm>. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

In Thumb assembly:

- outside an IT block, if ORRS <Rd>,<Rn>,<Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORRS <Rd>,<Rn> had been written
- inside an IT block, if ORR<c> <Rd>,<Rn>,<Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORR<c> <Rd>,<Rn> had been written.

To prevent either of these happening, use the .W qualifier.

The pre-UAL syntax ORR<c>S is equivalent to ORRS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR shifted;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ORR{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ
<shift>	<Rm> から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

Thumb アセンブラでは、次の規則が適用されます。

- IT ブロック外部で、ORRS <Rd>,<Rn>,<Rd> の <Rd> と <Rn> の両方が R0 ～ R7 の範囲に含まれる場合、ORRS <Rd>,<Rn> と記述されたものとして、エンコード T1 を使用してアセンブルされます。
- IT ブロック内部で、ORR<c> <Rd>,<Rn>,<Rd> の <Rd> と <Rn> の両方が R0 ～ R7 の範囲に含まれる場合、ORR<c> <Rd>,<Rn> と記述されたものとして、エンコード T1 を使用してアセンブルされます。

これらの事態を避けるには、.W 修飾子を使用します。

UAL 以前の構文の ORR<c>S は、ORRS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR shifted;
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.115 ORR (register-shifted register)

Bitwise OR (register-shifted register) performs a bitwise (inclusive) OR of a register value and a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ORR{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 1 0 0	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.115 ORR（レジスタによりシフトしたレジスタ）

ORR（ビット単位の論理和（レジスタによりシフトしたレジスタ））は、レジスタの値とレジスタによりシフトしたレジスタの値とのビット単位の（非排他的）論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ORR{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 1 0 0	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ORR{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The register that is shifted and used as the second operand.

<type> — The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR — Arithmetic shift right, encoded as type = 0b10
- LSL — Logical shift left, encoded as type = 0b00
- LSR — Logical shift right, encoded as type = 0b01
- ROR — Rotate right, encoded as type = 0b11.

<Rs> — The register whose bottom byte contains the amount to shift by.

The pre-UAL syntax ORR<c>S is equivalent to ORRS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ORR{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> — <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR — 算術右シフト。type = 0b10 としてエンコードされます。
- LSL — 論理左シフト。type = 0b00 としてエンコードされます。
- LSR — 論理右シフト。type = 0b01 としてエンコードされます。
- ROR — 右ローテート。type = 0b11 としてエンコードされます。

<Rs> — 最下位バイトにシフトのビット数を含むレジスタ

UAL 以前の構文の ORR<c>S は、ORRS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.116 PKH

Pack Halfword combines one halfword of its first operand with the other halfword of its shifted second operand.

Encoding T1 ARMv6T2, ARMv7

```
PKHBT<c> <Rd>,<Rn>,<Rm>{,LSL #<imm>}
PKHTB<c> <Rd>,<Rn>,<Rm>{,ASR #<imm>}
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12 11	10 9 8 7	6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 1 1 0	0	Rn	(0)	imm3	Rd	imm2	tb 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
PKHBT<c> <Rd>,<Rn>,<Rm>{,LSL #<imm>}
PKHTB<c> <Rd>,<Rn>,<Rm>{,ASR #<imm>}
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8 7	6	5 4	3 2 1 0
cond	0 1 1 0 1 0 0 0	Rn	Rd	imm5	tb	0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm5);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.116 PKH

PKH（ハーフワードのパッキング）は、最初のオペランドの 1 ハーフワードを、シフトされた 2 番目のオペランドの 1 ハーフワードと組み合わせます。

エンコード T1 ARMv6T2、ARMv7

```
PKHBT<c> <Rd>,<Rn>,<Rm>{,LSL #<imm>}
PKHTB<c> <Rd>,<Rn>,<Rm>{,ASR #<imm>}
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12 11	10 9 8 7	6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 1 1 0	0	Rn	(0)	imm3	Rd	imm2	tb 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
PKHBT<c> <Rd>,<Rn>,<Rm>{,LSL #<imm>}
PKHTB<c> <Rd>,<Rn>,<Rm>{,ASR #<imm>}
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8 7	6	5 4	3 2 1 0
cond	0 1 1 0 1 0 0 0	Rn	Rd	imm5	tb	0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm5);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
PKHBT<c><q>  {<Rd>,} <Rn>, <Rm> {, LSL #<imm>}        tbform == FALSE
PKHTB<c><q>  {<Rd>,} <Rn>, <Rm> {, ASR #<imm>}        tbform == TRUE
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rn>` The first operand register.

`<Rm>` The register that is optionally shifted and used as the second operand.

`<imm>` The shift to apply to the value read from `<Rm>`, encoded in imm3:imm2 for encoding T1 and imm5 for encoding A1.

For `PKHBT`, it is one of:

omitted No shift, encoded as `0b00000`

1-31 Left shift by specified number of bits, encoded as a binary number.

For `PKHTB`, it is one of:

omitted Instruction is a pseudo-instruction and is assembled as though `PKHBT<c><q> <Rd>,<Rm>,<Rn>` had been written

1-32 Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as `0b00000`. Other shift amounts are encoded as binary numbers.

Note

An assembler can permit `<imm>` = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = Shift(R[m], shift_t, shift_n, APSR.C);  // APSR.C ignored
    R[d]<15:0>  = if tbform then operand2<15:0> else R[n]<15:0>;
    R[d]<31:16> = if tbform then R[n]<31:16>    else operand2<31:16>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
PKHBT<c><q>  {<Rd>,} <Rn>, <Rm> {, LSL #<imm>}        tbform == FALSE
PKHTB<c><q>  {<Rd>,} <Rn>, <Rm> {, ASR #<imm>}        tbform == TRUE
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rn>` 最初のオペランドレジスタ

`<Rm>` オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ

`<imm>` `<Rm>` から読み出された値に適用されるシフト。エンコード T1 では imm3:imm2 にエンコードされ、エンコード A1 では imm5 にエンコードされます。

`PKHBT` では、次のいずれかを指定します。

省略 シフトが適用されず、`0b00000` としてエンコードされます。

1-31 指定したビット数だけ左シフトが適用され、バイナリ数値としてエンコードされます。

`PKHTB` では、次のいずれかを指定します。

省略 命令は擬似命令となり、`PKHBT<c><q> <Rd>,<Rm>,<Rn>` と記述されたものとしてアセンブルされます。

1-32 指定したビット数だけ算術右シフトが適用されます。32 ビットのシフトは、`0b00000` としてエンコードされます。その他のシフトは、バイナリ数値としてエンコードされます。

注

アセンブラは、シフトの省略と同じことを意味する `<imm>` = 0 を許可できます。しかし、これは標準的な UAL ではなく、逆アセンブルには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = Shift(R[m], shift_t, shift_n, APSR.C);  // APSR.C ignored
    R[d]<15:0>  = if tbform then operand2<15:0> else R[n]<15:0>;
    R[d]<31:16> = if tbform then R[n]<31:16>    else operand2<31:16>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.117 PLD, PLDW (immediate)

Preload Data signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache. For more information, see *Behavior of Preload Data (PLD, PLDW) and Preload Instruction (PLI) with caches* on page B2-7.

On an architecture variant that includes both the `PLD` and `PLDW` instructions, the `PLD` instruction signals that the likely memory access is a read, and the `PLDW` instruction signals that it is a write.

Encoding T1 ARMv6T2, ARMv7 for PLD
ARMv7 with MP Extensions for PLDW

```
PLD{W}<c> [<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	W	1	Rn				1	1	1	1	imm12											

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;  is_pldw = (W == '1');
```

Encoding T2 ARMv6T2, ARMv7 for PLD
ARMv7 with MP Extensions for PLDW

```
PLD{W}<c> [<Rn>,#-<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	W	1	Rn				1	1	1	1	1	1	0	0	imm8							

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;  is_pldw = (W == '1');
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7 for PLD
ARMv7 with MP Extensions for PLDW

```
PLD{W} [<Rn>,#+/-<imm12>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	U	R	0	1	Rn				(1)	(1)	(1)	(1)	imm12											

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');  is_pldw = (R == '0');
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.117 PLD、PLDW（イミディエート）

PLD、PLDW（データのプリロード（イミディエート））は、指定されたアドレスからのデータメモリ アクセスが近く発生する可能性があることをメモリシステムに伝達します。メモリシステムは、応答として、指定されたアドレスを含むキャッシュラインをデータキャッシュにプリロードするなど、メモリアクセスを高速化する動作を実行できます。詳細については、*キャッシュを使用したデータのプリロード (PLD、PLDW) と命令のプリロード (PLI) の動作* P. B2-7 を参照して下さい。

`PLD` 命令と `PLDW` 命令の両方を含むアーキテクチャバリアントでは、`PLD` 命令は、発生する可能性があるメモリアクセスが読み出しであることを伝達し、`PLDW` 命令は、それが書き込みであることを伝達します。

エンコード T1 ARMv6T2、ARMv7（PLD の場合）
MP 拡張機能を伴う ARMv7（PLDW の場合）

```
PLD{W}<c> [<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	W	1	Rn				1	1	1	1	imm12											

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;  is_pldw = (W == '1');
```

エンコード T2 ARMv6T2、ARMv7（PLD の場合）
MP 拡張機能を伴う ARMv7（PLDW の場合）

```
PLD{W}<c> [<Rn>,#-<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	W	1	Rn				1	1	1	1	1	1	0	0	imm8							

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;  is_pldw = (W == '1');
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7（PLD の場合）
MP 拡張機能を伴う ARMv7（PLDW の場合）

```
PLD{W} [<Rn>,#+/-<imm12>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	U	R	0	1	Rn				(1)	(1)	(1)	(1)	imm12											

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');  is_pldw = (R == '0');
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
PLD{W}<c><q>  [<Rn> {, #+/-<imm>}]
```

where:

W — If specified, selects PLDW, encoded as W = 1 in Thumb encodings and R = 0 in ARM encodings. If omitted, selects PLD, encoded as W = 0 in Thumb encodings and R = 1 in ARM encodings.

<c><q> — See *Standard assembler syntax fields* on page A8-7. An ARM PLD or PLDW instruction must be unconditional.

<Rn> — The base register. The SP can be used. For PC use in the PLD instruction, see *PLD (literal)* on page A8-238.

+/- — Is + or omitted to indicate that the immediate offset is added to the base register value (add == TRUE), or – to indicate that the offset is to be subtracted (add == FALSE). Different instructions are generated for #0 and #-0.

<imm> — The immediate offset used to form the address. This offset can be omitted, meaning an offset of 0. Values are:

Encoding T1, A1 any value in the range 0-4095

Encoding T2 any value in the range 0-255.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
PLD{W}<c><q>  [<Rn> {, #+/-<imm>}]
```

各項目の説明については以下を参照して下さい。

W — 指定した場合、PLDW が選択され、Thumb エンコードでは W = 1 として、ARM エンコードでは R = 0 としてそれぞれエンコードされます。省略した場合、PLD が選択され、Thumb エンコードでは W = 0 として、ARM エンコードでは R = 1 としてそれぞれエンコードされます。

<c><q> — *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。ARM PLD または PLDW は常に無条件命令です。

<Rn> — ベースレジスタ。このレジスタには SP を指定できます。PLD 命令での PC の指定については、*PLD（リテラル）* P. A8- 238 を参照して下さい。

+/- — イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、「+」にするか、省略します。減算する場合 (add == FALSE) は、「-」にします。#0 と #-0 では異なる命令が生成されます。

<imm> — アドレスを形成するために使用されるイミディエートオフセット。このオフセットは省略可能で、その場合オフセットは 0 と見なされます。値は次のとおりです。

エンコード T1、A1 0 ～ 4095 の任意の値

エンコード T2 0 ～ 255 の任意の値

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.118 PLD (literal)

Preload Data signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache. For more information, see *Behavior of Preload Data (PLD, PLDW) and Preload Instruction (PLI) with caches* on page B2-7.

Encoding T1 ARMv6T2, ARMv7

```
PLD<c> <label>
PLD<c> [PC,#-0]                                   Special case
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	U	0	(0)	1	1	1	1	1	1	1	1	1	imm12											

```
imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
PLD <label>
PLD [PC,#-0]                                      Special case
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	U	(1)	0	1	1	1	1	1	(1)	(1)	(1)	(1)	imm12											

```
imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.118 PLD（リテラル）

PLD（データのプリロード（リテラル））は、指定されたアドレスからのデータメモリ アクセスが近く発生する可能性があることをメモリシステムに伝達します。メモリシステムは、応答として、指定されたアドレスを含むキャッシュラインをデータキャッシュにプリロードするなど、メモリアクセスを高速化する動作を実行できます。詳細については、*キャッシュを使用したデータのプリロード (PLD、PLDW) と命令のプリロード (PLI) の動作* P. B2-7 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
PLD<c> <label>
PLD<c> [PC,#-0]                                   特殊なケース
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	U	0	(0)	1	1	1	1	1	1	1	1	1	imm12											

```
imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
PLD <label>
PLD [PC,#-0]                                      特殊なケース
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	U	(1)	0	1	1	1	1	1	(1)	(1)	(1)	(1)	imm12											

```
imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`PLD<c><q> <label>`	Normal form
`PLD<c><q> [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `PLD` instruction must be unconditional.

`<label>` The label of the literal data item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the `Align(PC,4)` value of this instruction to the label. The offset must be in the range –4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

`+/-` Is + or omitted to indicate that the immediate offset is added to the `Align(PC,4)` value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` The immediate offset used to form the address. Values are in the range 0-4095.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    Hint_PreloadData(address);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`PLD<c><q> <label>`	通常の形式
`PLD<c><q> [PC, #+/-<imm>]`	代替形式

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `PLD` は常に無条件命令です。

`<label>` 近いうちにアクセスされる可能性が高いリテラルデータ項目のラベル。アセンブラは、この命令の `Align(PC,4)` の値からこのラベルまでのオフセットに必要な値を計算します。オフセットに許容される値の範囲は – 4095 ～ 4095 です。

オフセットが 0 または正の場合、`imm32` はオフセットと等しく、`add == TRUE` になります。

オフセットが負の場合、`imm32` は負のオフセットと等しく、`add == FALSE` となります。

`+/-` イミディエートオフセットを `Align(PC,4)` の値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合 (`add == FALSE`) は、「−」にします。`.#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。値の範囲は 0 ～ 4095 です。

代替構文では、通常の構文では指定できない 0 の減算など、オフセットおよびイミディエートオフセットの加算または減算を個別に指定することが許可されます。詳細については、*UAL 命令構文でのラベルの使用* P. A4-5 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    Hint_PreloadData(address);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.119 PLD, PLDW (register)

Preload Data signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache. For more information, see *Behavior of Preload Data (PLD, PLDW) and Preload Instruction (PLI) with caches* on page B2-7.

On an architecture variant that includes both the `PLD` and `PLDW` instructions, the `PLD` instruction signals that the likely memory access is a read, and the `PLDW` instruction signals that it is a write.

Encoding T1 ARMv6T2, ARMv7 for PLD
ARMv7 with MP Extensions for PLDW

```
PLD{W}<c> [<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	W	1	Rn				1	1	1	1	0	0	0	0	0	0	imm2		Rm			

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;  is_pldw = (W == '1');
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7 for PLD
ARMv7 with MP Extensions for PLDW

```
PLD{W}<c> [<Rn>,+/-<Rm>{, <shift>}]
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	U	R	0	1	Rn				(1)	(1)	(1)	(1)	imm5					type		0	Rm			

```
n = UInt(Rn);  m = UInt(Rm);  add = (U == '1');  is_pldw = (R == '0');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.119 PLD、PLDW（レジスタ）

PLD、PLDW（データのプリロード（レジスタ））は、指定されたアドレスからのデータメモリアクセスが近く発生する可能性があることをメモリシステムに伝達します。メモリシステムは、応答として、指定されたアドレスを含むキャッシュラインをデータキャッシュにプリロードするなど、メモリアクセスを高速化する動作を実行できます。詳細については、*キャッシュを使用したデータのプリロード (PLD、PLDW) と命令のプリロード (PLI) の動作* P. B2-7 を参照して下さい。

`PLD` 命令と `PLDW` 命令の両方を含むアーキテクチャバリアントでは、`PLD` 命令は、発生する可能性があるメモリアクセスが読み出しであることを伝達し、`PLDW` 命令は、それが書き込みであることを伝達します。

エンコード T1 ARMv6T2、ARMv7（PLD の場合）
MP 拡張機能を伴う ARMv7（PLDW の場合）

```
PLD{W}<c> [<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	W	1	Rn				1	1	1	1	0	0	0	0	0	0	imm2		Rm			

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;  is_pldw = (W == '1');
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7（PLD の場合）
MP 拡張機能を伴う ARMv7（PLDW の場合）

```
PLD{W}<c> [<Rn>,+/-<Rm>{, <shift>}]
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	U	R	0	1	Rn				(1)	(1)	(1)	(1)	imm5					type		0	Rm			

```
n = UInt(Rn);  m = UInt(Rm);  add = (U == '1');  is_pldw = (R == '0');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
PLD[W]<c><q>  [<Rn>, +/-<Rm> {, <shift>}]
```

where:

W — If specified, selects PLDW, encoded as W = 1 in Thumb encodings and R = 0 in ARM encodings. If omitted, selects PLD, encoded as W = 0 in Thumb encodings and R = 1 in ARM encodings.

<c><q> — See *Standard assembler syntax fields* on page A8-7. An ARM PLD or PLDW instruction must be unconditional.

<Rn> — Is the base register. The SP can be used.

+/- — Is + or omitted if the optionally shifted value of <Rm> is to be added to the base register value (add == TRUE), or – if it is to be subtracted (permitted in ARM code only, add == FALSE).

<Rm> — Contains the offset that is optionally shifted and applied to the value of <Rn> to form the address.

<shift> — The shift to apply to the value read from <Rm>. If absent, no shift is applied. For encoding T1, <shift> can only be omitted, encoded as imm2 = 0b00, or LSL #<imm> with <imm> = 1, 2, or 3, with <imm> encoded in imm2. For encoding A1, see *Shifts applied to a register* on page A8-10.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
PLD[W]<c><q>  [<Rn>, +/-<Rm> {, <shift>}]
```

各項目の説明については以下を参照して下さい。

W — 指定した場合、PLDW が選択され、Thumb エンコードでは W = 1 として、ARM エンコードでは R = 0 としてそれぞれエンコードされます。省略した場合、PLD が選択され、Thumb エンコードでは W = 0 として、ARM エンコードでは R = 1 としてそれぞれエンコードされます。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM PLD または PLDW は常に無条件命令です。

<Rn> — ベースレジスタ。このレジスタには SP を指定できます。

+/- — オプションでシフトされる <Rm> の値をベースレジスタの値に加算する場合 (add == TRUE) は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、add == FALSE）は、「-」にします。

<Rm> — アドレス生成のためにオプションとしてシフトされ、<Rn> の値に適用されるオフセットを格納します。

<shift> — <Rm> から読み出された値に適用されるシフト。省略した場合、シフトは適用されません。エンコード T1 では、<shift> として省略（imm2 = 0b00 としてエンコードされます）、または LSL #<imm>（<imm> = 1、2、3 のいずれか、<imm> は imm2 にエンコードされます）のみが使用できます。エンコード A1 については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.120 PLI (immediate, literal)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache. For more information, see *Behavior of Preload Data (PLD, PLDW) and Preload Instruction (PLI) with caches* on page B2-7.

Encoding T1 ARMv7

```
PLI<c> [<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	0	0	1	Rn				1	1	1	1	imm12											

```
if Rn == '1111' then SEE encoding T3;
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;
```

Encoding T2 ARMv7

```
PLI<c> [<Rn>,#-<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	Rn				1	1	1	1	1	1	0	0	imm8							

```
if Rn == '1111' then SEE encoding T3;
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;
```

Encoding T3 ARMv7

```
PLI<c> <label>
PLI<c> [PC,#-0]                                   Special case
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	U	0	0	1	1	1	1	1	1	1	1	1	imm12											

```
n = 15;  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

Encoding A1 ARMv7

```
PLI [<Rn>,#+/-<imm12>]
PLI <label>
PLI [PC,#-0]                                      Special case
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	U	1	0	1	Rn				(1)	(1)	(1)	(1)	imm12											

```
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.120 PLI（イミディエート、リテラル）

PLI（命令のプリロード（イミディエート、リテラル））は、指定されたアドレスからの命令メモリ アクセスが近く発生する可能性があることをメモリシステムに伝達します。メモリシステムは、応答として、指定されたアドレスを含むキャッシュラインを命令キャッシュにプリロードするなど、メモリアクセスを高速化する動作を実行できます。詳細については、*キャッシュを使用したデータのプリロード (PLD、PLDW) と命令のプリロード (PLI) の動作* P. B2-7 を参照して下さい。

エンコード T1 ARMv7

```
PLI<c> [<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	0	0	1	Rn				1	1	1	1	imm12											

```
if Rn == '1111' then SEE encoding T3;
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;
```

エンコード T2 ARMv7

```
PLI<c> [<Rn>,#-<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	Rn				1	1	1	1	1	1	0	0	imm8							

```
if Rn == '1111' then SEE encoding T3;
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;
```

エンコード T3 ARMv7

```
PLI<c> <label>
PLI<c> [PC,#-0]                                   特殊なケース
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	U	0	0	1	1	1	1	1	1	1	1	1	imm12											

```
n = 15;  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

エンコード A1 ARMv7

```
PLI [<Rn>,#+/-<imm12>]
PLI <label>
PLI [PC,#-0]                                      特殊なケース
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	U	1	0	1	Rn				(1)	(1)	(1)	(1)	imm12											

```
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`PLI<c><q> [<Rn> {, #+/-<imm>}]`	Immediate form
`PLI<c><q> <label>`	Normal literal form
`PLI<c><q> [PC, #+/-<imm>]`	Alternative literal form

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `PLI` instruction must be unconditional.

`<Rn>` Is the base register. The SP can be used.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` The immediate offset used to form the address. For the immediate form of the syntax, `<imm>` can be omitted, in which case the `#0` form of the instruction is assembled. Values are:

Encoding T1, T3, A1 any value in the range 0 to 4095

Encoding T2 any value in the range 0 to 255.

`<label>` The label of the instruction that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the `Align(PC,4)` value of this instruction to the label. The offset must be in the range –4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

For the literal forms of the instruction, encoding T3 is used, or Rn is encoded as '1111' in encoding A1, to indicate that the PC is the base register.

The alternative literal syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`PLI<c><q> [<Rn> {, #+/-<imm>}]`	イミディエート形式
`PLI<c><q> <label>`	通常のリテラル形式
`PLI<c><q> [PC, #+/-<imm>]`	代替のリテラル形式

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `PLI` は常に無条件命令です。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合 (`add == FALSE`) は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。イミディエート形式の構文では、`<imm>` を省略できます。この場合、`#0` 形式の命令がアセンブルされます。値は次のとおりです。

エンコード T1、T3、A1 0 ～ 4095 の任意の値

エンコード T2 0 ～ 255 の任意の値

`<label>` 近いうちにアクセスされる可能性が高い命令のラベル。アセンブラは、この命令の `Align(PC,4)` の値からこのラベルまでのオフセットに必要な値を計算します。オフセットに許容される値の範囲は – 4095 ～ 4095 です。

オフセットが 0 または正の場合、`imm32` はオフセットと等しく、`add == TRUE` になります。

オフセットが負の場合、`imm32` は負のオフセットと等しく、`add == FALSE` になります。

リテラル形式の命令では、エンコード T3 が使用されるか、またはエンコード A1 で Rn が '1111' としてエンコードされ、PC がベースレジスタであることが示されます。

代替のリテラル構文では、通常の構文では指定できない 0 の減算など、オフセットおよびイミディエートオフセットの加算または減算を個別に指定することが許可されます。詳細については、*UAL 命令構文でのラベルの使用* P. A4-5 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.121 PLI (register)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache. For more information, see *Behavior of Preload Data (PLD, PLDW) and Preload Instruction (PLI) with caches* on page B2-7.

Encoding T1 ARMv7

```
PLI<c> [<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	Rn				1	1	1	1	0	0	0	0	0	0	imm2		Rm			

```
if Rn == '1111' then SEE PLI (immediate, literal);
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv7

```
PLI [<Rn>,+/-<Rm>{, <shift>}]
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	0	U	1	0	1	Rn				(1)	(1)	(1)	(1)	imm5					type		0	Rm			

```
n = UInt(Rn);  m = UInt(Rm);  add = (U == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.121 PLI（レジスタ）

PLI（命令のプリロード（レジスタ））は、指定されたアドレスからの命令メモリ アクセスが近く発生する可能性があることをメモリシステムに伝達します。メモリシステムは、応答として、指定されたアドレスを含むキャッシュラインを命令キャッシュにプリロードするなど、メモリアクセスを高速化する動作を実行できます。詳細については、*キャッシュを使用したデータのプリロード (PLD、PLDW) と命令のプリロード (PLI) の動作* P. B2-7 を参照して下さい。

エンコード T1 ARMv7

```
PLI<c> [<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	Rn				1	1	1	1	0	0	0	0	0	0	imm2		Rm			

```
if Rn == '1111' then SEE PLI (immediate, literal);
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv7

```
PLI [<Rn>,+/-<Rm>{, <shift>}]
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	0	U	1	0	1	Rn				(1)	(1)	(1)	(1)	imm5					type		0	Rm			

```
n = UInt(Rn);  m = UInt(Rm);  add = (U == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
PLI<c><q>  [<Rn>, +/-<Rm> {, <shift>}]
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7. An ARM PLI instruction must be unconditional.

<Rn> Is the base register. The SP can be used.

+/- Is + or omitted if the optionally shifted value of <Rm> is to be added to the base register value (add == TRUE), or – if it is to be subtracted (permitted in ARM code only, add == FALSE).

<Rm> Contains the offset that is optionally shifted and applied to the value of <Rn> to form the address.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. For encoding T1, <shift> can only be omitted, encoded as imm2 = 0b00, or LSL #<imm> with <imm> = 1, 2, or 3, with <imm> encoded in imm2. For encoding A1, see *Shifts applied to a register* on page A8-10.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadInstr(address);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
PLI<c><q>  [<Rn>, +/-<Rm> {, <shift>}]
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM PLI は常に無条件命令です。

<Rn> ベースレジスタ。このレジスタには SP を指定できます。

+/- オプションでシフトされる <Rm> の値をベースレジスタの値に加算する場合 (add == TRUE) は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、add == FALSE）は、「-」にします。

<Rm> アドレス生成のためにオプションとしてシフトされ、<Rn> の値に適用されるオフセットを格納します。

<shift> <Rm> から読み出された値に適用されるシフト。省略した場合、シフトは適用されません。エンコード T1 では、<shift> として省略（imm2 = 0b00 としてエンコードされます）、または LSL #<imm>（<imm> = 1、2、3 のいずれか、<imm> は imm2 にエンコードされます）のみが使用できます。エンコード A1 については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadInstr(address);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.122 POP

Pop Multiple Registers loads multiple registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
POP<c> <registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	0	P	register_list							

```
registers = P:'0000000':register_list;  if BitCount(registers) < 1 then UNPREDICTABLE;
```

Encoding T2 ARMv6T2, ARMv7

```
POP<c>.W <registers>
```

<registers> contains more than one register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	1	1	1	1	0	1	P	M	(0)	register_list												

```
registers = P:M:'0':register_list;
if BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

```
POP<c>.W <registers>
```

<registers> contains one register, <Rt>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	1	1	1	0	1	Rt				1	0	1	1	0	0	0	0	0	1	0	0

```
t = UInt(Rt);  registers = Zeros(16);  registers<t> = '1';
if t == 13 || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
POP<c> <registers>
```

<registers> contains more than one register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	0	0	1	0	1	1	1	1	0	1	register_list															

```
if BitCount(register_list) < 2 then SEE LDM / LDMIA / LDMFD;
registers = register_list;
if registers<13> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```

Encoding A2 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
POP<c> <registers>
```

<registers> contains one register, <Rt>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	0	0	1	0	0	1	1	1	0	1	Rt				0	0	0	0	0	0	0	0	0	1	0	0

```
t = UInt(Rt);  registers = Zeros(16);  registers<t> = '1';
if t == 13 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.122 POP

POP（複数レジスタポップ）は、複数のレジスタをスタックからロード（SP のアドレスから開始される、連続するメモリ位置からロード）し、ロードされたデータの次のアドレスを指し示すように SP を更新します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
POP<c> <registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	0	P	register_list							

```
registers = P:'0000000':register_list;  if BitCount(registers) < 1 then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

```
POP<c>.W <registers>
```

<registers> には複数のレジスタが含まれる

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	1	1	1	1	0	1	P	M	(0)	register_list												

```
registers = P:M:'0':register_list;
if BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

```
POP<c>.W <registers>
```

<registers> には 1 つのレジスタ <Rt> が含まれる

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	1	1	1	0	1	Rt				1	0	1	1	0	0	0	0	0	1	0	0

```
t = UInt(Rt);  registers = Zeros(16);  registers<t> = '1';
if t == 13 || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
POP<c> <registers>
```

<registers> には複数のレジスタが含まれる

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	0	0	1	0	1	1	1	1	0	1	register_list															

```
if BitCount(register_list) < 2 then SEE LDM / LDMIA / LDMFD;
registers = register_list;
if registers<13> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```

エンコード A2 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
POP<c> <registers>
```

<registers> には 1 つのレジスタ <Rt> が含まれる

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	0	0	1	0	0	1	1	1	0	1	Rt				0	0	0	0	0	0	0	0	0	1	0	0

```
t = UInt(Rt);  registers = Zeros(16);  registers<t> = '1';
if t == 13 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`POP<c><q> <registers>`	Standard syntax
`LDM<c><q> SP!, <registers>`	Equivalent `LDM` syntax

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<registers>` Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address.

If the list contains more than one register, the instruction is assembled to encoding T1, T2, or A1. If the list contains exactly one register, the instruction is assembled to encoding T1, T3, or A2.

The SP can only be in the list in ARM code before ARMv7. ARM instructions that include the SP in the list are deprecated, and the value of the SP after such an instruction is `UNKNOWN`.

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. In ARMv5T and above, this is an interworking branch, see *Pseudocode details of operations on ARM core registers* on page A2-12. In Thumb code, if the PC is in the list:

- the LR must not be in the list
- the instruction must be either outside any IT block, or the last instruction in an IT block.

ARM instructions that include both the LR and the PC in the list are deprecated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(13);
    address = SP;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i} = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if registers<13> == '0' then SP = SP + 4*BitCount(registers);
    if registers<13> == '1' then SP = bits(32) UNKNOWN;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`POP<c><q> <registers>`	標準構文
`LDM<c><q> SP!, <registers>`	等価な `LDM` 構文

各項目の説明については以下を参照して下さい。

`<c><q>` 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<registers>` ロードされる 1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリスト。最下位のメモリアドレスに置かれているデータは最も小さい番号のレジスタにロードされ、以下順に、最上位のメモリアドレスのデータは最も大きい番号のレジスタにロードされます。

リストに複数のレジスタが含まれる場合、命令はエンコード T1、T2、または A1 にアセンブルされます。リストに含まれるレジスタが 1 つのみである場合、命令はエンコード T1、T3、または A2 にアセンブルされます。

ARMv7 以前では、ARM コードでのみリストに SP を含めることができます。リストに SP を含める ARM 命令は推奨されません。このような命令の後の SP の値は未知です。

リストに PC を含めることができます。リストに PC が含まれている場合、PC にロードされるアドレスに命令が分岐します。ARMv5T およびそれ以降では、これはインターワーキング分岐です。詳細については、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 を参照して下さい。Thumb コードでは、リストに PC が含まれている場合に次の規則が適用されます。

- LR をリストに含めることはできません。
- この命令は、IT ブロック外部に配置するか、IT ブロック内部の最後の命令にする必要があります。

リストに LR と PC の両方を含める ARM 命令は推奨されません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(13);
    address = SP;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i} = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
    if registers<13> == '0' then SP = SP + 4*BitCount(registers);
    if registers<13> == '1' then SP = bits(32) UNKNOWN;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.123 PUSH

Push Multiple Registers stores multiple registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
PUSH<c> <registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	M	register_list							

```
registers = '0':M:'000000':register_list;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Encoding T2 ARMv6T2, ARMv7

```
PUSH<c>.W <registers>
```

<registers> contains more than one register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	1	0	1	1	0	1	(0)	M	(0)	register_list												

```
registers = '0':M:'0':register_list;
if BitCount(registers) < 2 then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

```
PUSH<c>.W <registers>
```

<registers> contains one register, <Rt>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	0	1	1	0	1	Rt				1	1	0	1	0	0	0	0	0	1	0	0

```
t = UInt(Rt);  registers = Zeros(16);  registers<t> = '1';
if BadReg(t) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
PUSH<c> <registers>
```

<registers> contains more than one register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	0	1	0	0	1	0	1	1	0	1	register_list															

```
if BitCount(register_list) < 2 then SEE STMDB / STMFD;
registers = register_list;
```

Encoding A2 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
PUSH<c> <registers>
```

<registers> contains one register, <Rt>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	0	1	0	0	1	0	1	1	0	1	Rt				0	0	0	0	0	0	0	0	0	1	0	0

```
t = UInt(Rt);  registers = Zeros(16);  registers<t> = '1';
if t == 13 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.123 PUSH

PUSH（複数レジスタプッシュ）は、複数のレジスタをスタックにストア（SP のアドレスの 1 つ前で終了する、連続するメモリ位置にストア）し、ストアされたデータの開始アドレスを指し示すように SP を更新します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
PUSH<c> <registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	M	register_list							

```
registers = '0':M:'000000':register_list;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

```
PUSH<c>.W <registers>
```

<registers> には複数のレジスタが含まれる

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	1	0	1	1	0	1	(0)	M	(0)	register_list												

```
registers = '0':M:'0':register_list;
if BitCount(registers) < 2 then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

```
PUSH<c>.W <registers>
```

<registers> には 1 つのレジスタ <Rt> が含まれる

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	0	1	1	0	1	Rt				1	1	0	1	0	0	0	0	0	1	0	0

```
t = UInt(Rt);  registers = Zeros(16);  registers<t> = '1';
if BadReg(t) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
PUSH<c> <registers>
```

<registers> には複数のレジスタが含まれる

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	0	1	0	0	1	0	1	1	0	1	register_list															

```
if BitCount(register_list) < 2 then SEE STMDB / STMFD;
registers = register_list;
```

エンコード A2 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
PUSH<c> <registers>
```

<registers> には 1 つのレジスタ <Rt> が含まれる

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	0	1	0	0	1	0	1	1	0	1	Rt				0	0	0	0	0	0	0	0	0	1	0	0

```
t = UInt(Rt);  registers = Zeros(16);  registers<t> = '1';
if t == 13 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`PUSH<c><q> <registers>`	Standard syntax
`STMDB<c><q> SP!, <registers>`	Equivalent STM syntax

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<registers>` Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address.

If the list contains more than one register, the instruction is assembled to encoding T1, T2, or A1. If the list contains exactly one register, the instruction is assembled to encoding T1, T3, or A2.

The SP and PC can be in the list in ARM code, but not in Thumb code. However, ARM instructions that include the SP or the PC in the list are deprecated, and if the SP is in the list, the value the instruction stores for the SP is UNKNOWN.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(13);
    address = SP - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == 13 && i != LowestSetBit(registers) then  // Only possible for encoding A1
                MemA[address,4] = bits(32) UNKNOWN;
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1 or A2
        MemA[address,4] = PCStoreValue();
    SP = SP - 4*BitCount(registers);
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`PUSH<c><q> <registers>`	標準構文
`STMDB<c><q> SP!, <registers>`	等価な STM 構文

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<registers>` ストアされる 1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリスト。番号が最も小さいレジスタが最下位のメモリアドレスにストアされ、以下順にストアが行われ、番号が最も大きいレジスタが最上位のメモリアドレスにストアされます。

リストに複数のレジスタが含まれる場合、命令はエンコード T1、T2、または A1 にアセンブルされます。リストに含まれるレジスタが 1 つのみである場合、命令はエンコード T1、T3、または A2 にアセンブルされます。

ARM コードではリストに SP と PC を含めることができますが、Thumb コードではできません。しかし、リストに SP または PC を含める ARM 命令は推奨されません。リストに SP が含まれている場合は、SP に関して命令がストアする値は未知です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(13);
    address = SP - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == 13 && i != LowestSetBit(registers) then  // Only possible for encoding A1
                MemA[address,4] = bits(32) UNKNOWN;
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1 or A2
        MemA[address,4] = PCStoreValue();
    SP = SP - 4*BitCount(registers);
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.124 QADD

Saturating Add adds two register values, saturates the result to the 32-bit signed integer range $-2^{31} \le x \le 2^{31} - 1$, and writes the result to the destination register. If saturation occurs, it sets the Q flag in the APSR.

Encoding T1 ARMv6T2, ARMv7

```
QADD<c> <Rd>,<Rm>,<Rn>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	1	0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
QADD<c> <Rd>,<Rm>,<Rn>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	0 0 0 0	Rn	Rd	(0) (0) (0) (0)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.124 QADD

QADD（飽和加算）は、2 つのレジスタ値を加算し、結果を符号付き 32 ビット整数の範囲 ($-2^{31} \leqq x \leqq 2^{31} - 1$) に飽和させ、結果をデスティネーションレジスタに書き込みます。飽和が発生すると、APSR の Q フラグがセットされます。

エンコード T1 ARMv6T2、ARMv7

```
QADD<c> <Rd>,<Rm>,<Rn>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	1	0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
QADD<c> <Rd>,<Rm>,<Rn>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	0 0 0 0	Rn	Rd	(0) (0) (0) (0)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
QADD<c><q>  {<Rd>,} <Rm>, <Rn>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rm>`	The first operand register.
`<Rn>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (R[d], sat) = SignedSatQ(SInt(R[m]) + SInt(R[n]), 32);
    if sat then
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
QADD<c><q>  {<Rd>,} <Rm>, <Rn>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rm>`	最初のオペランドレジスタ
`<Rn>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (R[d], sat) = SignedSatQ(SInt(R[m]) + SInt(R[n]), 32);
    if sat then
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.125 QADD16

Saturating Add 16 performs two 16-bit integer additions, saturates the results to the 16-bit signed integer range $-2^{15} \le x \le 2^{15} - 1$, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
QADD16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
QADD16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 0 0 1 0	Rn	Rd	(1) (1) (1) (1)	0 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.125 QADD16

QADD16（飽和加算 16）は、2 つの 16 ビット整数加算を実行し、結果を符号付き 16 ビット整数の範囲 $(-2^{15} \leqq x \leqq 2^{15} - 1)$ に飽和させ、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
QADD16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
QADD16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 0 0 1 0	Rn	Rd	(1) (1) (1) (1)	0 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
QADD16<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0>  = SignedSat(sum1, 16);
    R[d]<31:16> = SignedSat(sum2, 16);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
QADD16<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0>  = SignedSat(sum1, 16);
    R[d]<31:16> = SignedSat(sum2, 16);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.126 QADD8

Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range $-2^7 \le x \le 2^7 - 1$, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
QADD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
QADD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 0 1	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.126 QADD8

QADD8（飽和加算 8）は、4 つの 8 ビット整数加算を実行し、結果を符号付き 8 ビット整数の範囲 $(-2^7 \leqq x \leqq 2^7 - 1)$ に飽和させ、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
QADD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
QADD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 0 1	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
QADD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0>   = SignedSat(sum1, 8);
    R[d]<15:8>  = SignedSat(sum2, 8);
    R[d]<23:16> = SignedSat(sum3, 8);
    R[d]<31:24> = SignedSat(sum4, 8);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
QADD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0>   = SignedSat(sum1, 8);
    R[d]<15:8>  = SignedSat(sum2, 8);
    R[d]<23:16> = SignedSat(sum3, 8);
    R[d]<31:24> = SignedSat(sum4, 8);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.127 QASX

Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, saturates the results to the 16-bit signed integer range $-2^{15} \le x \le 2^{15} - 1$, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
QASX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
QASX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 0 0 1 0	Rn	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.127 QASX

QASX（交換付き飽和加算および減算）は、2 番目のオペランドの 2 つのハーフワードを交換し、1 つの 16 ビット整数加算と 1 つの 16 ビット減算を実行し、結果を 16 ビット符号付き整数の範囲 (- 2^{15} ≦ x ≦ 2^{15} - 1) に飽和させ、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
QASX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
QASX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 0 0 1 0	Rn	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
QASX<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<Rm> The second operand register.

The pre-UAL syntax QADDSUBX<c> is equivalent to QASX<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>  = SignedSat(diff, 16);
    R[d]<31:16> = SignedSat(sum, 16);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
QASX<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<Rm> 2 番目のオペランドレジスタ

UAL 以前の構文の QADDSUBX<c> は、QASX<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>  = SignedSat(diff, 16);
    R[d]<31:16> = SignedSat(sum, 16);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.128 QDADD

Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range $-2^{31} \le x \le 2^{31} - 1$. If saturation occurs in either operation, it sets the Q flag in the APSR.

Encoding T1 ARMv6T2, ARMv7

```
QDADD<c> <Rd>,<Rm>,<Rn>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	1	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
QDADD<c> <Rd>,<Rm>,<Rn>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	0 1 0 0	Rn	Rd	(0) (0) (0) (0)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.128 QDADD

QDADD（2 倍飽和加算）は、2 倍にしたレジスタの値を別のレジスタの値に加算し、結果をデスティネーションレジスタに書き込みます。2 倍と加算の両方の結果が符号付き 32 ビット整数の範囲 ($-2^{31} \leqq x \leqq 2^{31} - 1$) に飽和されます。いずれかの演算で飽和が発生すると、APSR の Q フラグがセットされます。

エンコード T1 ARMv6T2、ARMv7

```
QDADD<c> <Rd>,<Rm>,<Rn>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	1	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
QDADD<c> <Rd>,<Rm>,<Rn>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	0 1 0 0	Rn	Rd	(0) (0) (0) (0)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
QDADD<c><q>  {<Rd>,} <Rm>, <Rn>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rm>`	The first operand register.
`<Rn>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
    (R[d], sat2)  = SignedSatQ(SInt(R[m]) + SInt(doubled), 32);
    if sat1 || sat2 then
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
QDADD<c><q>  {<Rd>,} <Rm>, <Rn>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rm>`	最初のオペランドレジスタ
`<Rn>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
    (R[d], sat2)  = SignedSatQ(SInt(R[m]) + SInt(doubled), 32);
    if sat1 || sat2 then
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.129 QDSUB

Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range $-2^{31} \le x \le 2^{31} - 1$. If saturation occurs in either operation, it sets the Q flag in the APSR.

Encoding T1 ARMv6T2, ARMv7

```
QDSUB<c> <Rd>,<Rm>,<Rn>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	1	0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
QDSUB<c> <Rd>,<Rm>,<Rn>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	0 1 1 0	Rn	Rd	(0) (0) (0) (0)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.129 QDSUB

QDSUB（2 倍飽和減算）は、2 倍にしたレジスタの値を別のレジスタの値から減算し、結果をデスティネーションレジスタに書き込みます。2 倍と減算の両方の結果が符号付き 32 ビット整数の範囲 $(-2^{31} \leqq x \leqq 2^{31} - 1)$ に飽和されます。いずれかの演算で飽和が発生すると、APSR の Q フラグがセットされます。

エンコード T1 ARMv6T2、ARMv7

```
QDSUB<c> <Rd>,<Rm>,<Rn>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	1	0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
QDSUB<c> <Rd>,<Rm>,<Rn>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	0 1 1 0	Rn	Rd	(0) (0) (0) (0)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
QDSUB<c><q>  {<Rd>,} <Rm>, <Rn>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rm>`	The first operand register.
`<Rn>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
    (R[d], sat2)  = SignedSatQ(SInt(R[m]) - SInt(doubled), 32);
    if sat1 || sat2 then
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
QDSUB<c><q>  {<Rd>,} <Rm>, <Rn>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rm>`	最初のオペランドレジスタ
`<Rn>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
    (R[d], sat2)  = SignedSatQ(SInt(R[m]) - SInt(doubled), 32);
    if sat1 || sat2 then
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.130 QSAX

Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, saturates the results to the 16-bit signed integer range $-2^{15} \le x \le 2^{15} - 1$, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
QSAX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 1 0	Rn	1 1 1 1	Rd	0	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
QSAX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 0	Rn	Rd	(1) (1) (1) (1)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.130 QSAX

QSAX（交換付き飽和減算および加算）は、2 番目のオペランドの 2 つのハーフワードを交換し、1 つの 16 ビット整数減算と 1 つの 16 ビット加算を実行し、結果を 16 ビット符号付き整数の範囲 (- 2^{15} ≦ x ≦ 2^{15} - 1) に飽和させ、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
QSAX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 1 0	Rn	1 1 1 1	Rd	0	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
QSAX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 0	Rn	Rd	(1) (1) (1) (1)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
QSAX<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<Rm> The second operand register.

The pre-UAL syntax QSUBADDX<c> is equivalent to QSAX<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>  = SignedSat(diff, 16);
    R[d]<31:16> = SignedSat(sum, 16);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
QSAX<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<Rm> 2 番目のオペランドレジスタ

UAL 以前の構文の QSUBADDX<c> は、QSAX<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>  = SignedSat(diff, 16);
    R[d]<31:16> = SignedSat(sum, 16);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.131 QSUB

Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range $-2^{31} \le x \le 2^{31} - 1$, and writes the result to the destination register. If saturation occurs, it sets the Q flag in the APSR.

Encoding T1 ARMv6T2, ARMv7

```
QSUB<c> <Rd>,<Rm>,<Rn>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	1	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
QSUB<c> <Rd>,<Rm>,<Rn>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	0 0 1 0	Rn	Rd	(0) (0) (0) (0)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.131 QSUB

QSUB（飽和減算）は、1 つのレジスタの値を別のレジスタの値から減算し、結果を符号付き 32 ビット整数の範囲 $(-2^{31} \leqq x \leqq 2^{31} - 1)$ に飽和させ、結果をデスティネーションレジスタに書き込みます。飽和が発生すると、APSR の Q フラグがセットされます。

エンコード T1 ARMv6T2、ARMv7

```
QSUB<c> <Rd>,<Rm>,<Rn>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	1	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
QSUB<c> <Rd>,<Rm>,<Rn>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	0 0 1 0	Rn	Rd	(0) (0) (0) (0)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
QSUB<c><q>  {<Rd>,} <Rm>, <Rn>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rm>`	The first operand register.
`<Rn>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (R[d], sat) = SignedSatQ(SInt(R[m]) - SInt(R[n]), 32);
    if sat then
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
QSUB<c><q>  {<Rd>,} <Rm>, <Rn>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rm>`	最初のオペランドレジスタ
`<Rn>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (R[d], sat) = SignedSatQ(SInt(R[m]) - SInt(R[n]), 32);
    if sat then
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.132 QSUB16

Saturating Subtract 16 performs two 16-bit integer subtractions, saturates the results to the 16-bit signed integer range $-2^{15} \le x \le 2^{15} - 1$, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
QSUB16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
QSUB16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 0	Rn	Rd	(1) (1) (1) (1)	0 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.132 QSUB16

QSUB16（飽和減算 16）は、2 つの 16 ビット整数減算を実行し、結果を符号付き 16 ビット整数の範囲 $(-2^{15} \leqq x \leqq 2^{15} - 1)$ に飽和させ、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
QSUB16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
QSUB16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 0	Rn	Rd	(1) (1) (1) (1)	0 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
QSUB16<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0>  = SignedSat(diff1, 16);
    R[d]<31:16> = SignedSat(diff2, 16);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
QSUB16<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0>  = SignedSat(diff1, 16);
    R[d]<31:16> = SignedSat(diff2, 16);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.133 QSUB8

Saturating Subtract 8 performs four 8-bit integer subtractions, saturates the results to the 8-bit signed integer range $-2^7 \le x \le 2^7 - 1$, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
QSUB8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 0	Rn	1 1 1 1	Rd	0	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
QSUB8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 0	Rn	Rd	(1) (1) (1) (1)	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.133 QSUB8

QSUB8（飽和減算 8）は、4 つの 8 ビット整数減算を実行し、結果を符号付き 8 ビット整数の範囲 $(-2^7 \leqq x \leqq 2^7 - 1)$ に飽和させ、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
QSUB8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 0	Rn	1 1 1 1	Rd	0	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
QSUB8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 0	Rn	Rd	(1) (1) (1) (1)	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
QSUB8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0>   = SignedSat(diff1, 8);
    R[d]<15:8>  = SignedSat(diff2, 8);
    R[d]<23:16> = SignedSat(diff3, 8);
    R[d]<31:24> = SignedSat(diff4, 8);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
QSUB8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0>   = SignedSat(diff1, 8);
    R[d]<15:8>  = SignedSat(diff2, 8);
    R[d]<23:16> = SignedSat(diff3, 8);
    R[d]<31:24> = SignedSat(diff4, 8);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.134 RBIT

Reverse Bits reverses the bit order in a 32-bit register.

Encoding T1 ARMv6T2, ARMv7

```
RBIT<c> <Rd>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 1	Rm	1 1 1 1	Rd	1	0 1 0	Rm

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6T2, ARMv7

```
RBIT<c> <Rd>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	1 1 1 1	(1) (1) (1) (1)	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.134 RBIT

RBIT（ビット反転）は、32 ビットレジスタのビット順序を反転します。

エンコード T1 ARMv6T2、ARMv7

```
RBIT<c> <Rd>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 1	Rm	1 1 1 1	Rd	1	0 1 0	Rm

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6T2、ARMv7

```
RBIT<c> <Rd>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	1 1 1 1	(1) (1) (1) (1)	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
RBIT<c><q>  <Rd>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rm> The register that contains the operand. In encoding T1, its number must be encoded twice.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    for i = 0 to 31 do
        result<31-i> = R[m]<i>;
    R[d] = result;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
RBIT<c><q>  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rm> オペランドを含むレジスタ。エンコード T1 では、Rm の番号を 2 回エンコードする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    for i = 0 to 31 do
        result<31-i> = R[m]<i>;
    R[d] = result;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.135 REV

Byte-Reverse Word reverses the byte order in a 32-bit register.

Encoding T1 ARMv6*, ARMv7

```
REV<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	0	0	Rm				Rd	

```
d = UInt(Rd);  m = UInt(Rm);
```

Encoding T2 ARMv6T2, ARMv7

```
REV<c>.W <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rm				1	1	1	1	Rd				1	0	0	0	Rm			

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
REV<c> <Rd>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	1	(1)	(1)	(1)	(1)	Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.135 REV

REV（ワードのバイト反転）は、32 ビットレジスタのバイト順序を反転します。

エンコード T1 ARMv6*、ARMv7

```
REV<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	0	0	Rm				Rd	

```
d = UInt(Rd);  m = UInt(Rm);
```

エンコード T2 ARMv6T2、ARMv7

```
REV<c>.W <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rm				1	1	1	1	Rd				1	0	0	0	Rm			

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
REV<c> <Rd>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	1	(1)	(1)	(1)	(1)	Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
REV<c><q>  <Rd>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rm> The register that contains the operand. Its number must be encoded twice in encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<7:0>;
    result<23:16> = R[m]<15:8>;
    result<15:8>  = R[m]<23:16>;
    result<7:0>   = R[m]<31:24>;
    R[d] = result;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
REV<c><q>  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rm> オペランドを含むレジスタ。エンコード T2 では、Rm の番号を 2 回エンコードする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<7:0>;
    result<23:16> = R[m]<15:8>;
    result<15:8>  = R[m]<23:16>;
    result<7:0>   = R[m]<31:24>;
    R[d] = result;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.136 REV16

Byte-Reverse Packed Halfword reverses the byte order in each16-bit halfword of a 32-bit register.

Encoding T1 ARMv6*, ARMv7

```
REV16<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	0	1	Rm				Rd	

```
d = UInt(Rd);  m = UInt(Rm);
```

Encoding T2 ARMv6T2, ARMv7

```
REV16<c>.W <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rm				1	1	1	1	Rd				1	0	0	1	Rm			

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
REV16<c> <Rd>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	1	(1)	(1)	(1)	(1)	Rd				(1)	(1)	(1)	(1)	1	0	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.136 REV16

REV16（パッキングされたハーフワードのバイト反転）は、32 ビットレジスタの各 16 ビットハーフワードのバイト順序を反転します。

エンコード T1 ARMv6*、ARMv7

```
REV16<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	0	1	Rm				Rd	

```
d = UInt(Rd);  m = UInt(Rm);
```

エンコード T2 ARMv6T2、ARMv7

```
REV16<c>.W <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rm				1	1	1	1	Rd				1	0	0	1	Rm			

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
REV16<c> <Rd>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	1	(1)	(1)	(1)	(1)	Rd				(1)	(1)	(1)	(1)	1	0	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
REV16<c><q>  <Rd>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rm> The register that contains the operand. Its number must be encoded twice in encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<23:16>;
    result<23:16> = R[m]<31:24>;
    result<15:8>  = R[m]<7:0>;
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
REV16<c><q>  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rm> オペランドを含むレジスタ。エンコード T2 では、Rm の番号を 2 回エンコードする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<23:16>;
    result<23:16> = R[m]<31:24>;
    result<15:8>  = R[m]<7:0>;
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.137 REVSH

Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign-extends the result to 32 bits.

Encoding T1 ARMv6*, ARMv7

```
REVSH<c> <Rd>,<Rm>
```

15 14 13 12	11 10 9 8	7 6	5 4 3	2 1 0
1 0 1 1	1 0 1 0	1 1	Rm	Rd

```
d = UInt(Rd);  m = UInt(Rm);
```

Encoding T2 ARMv6T2, ARMv7

```
REVSH<c>.W <Rd>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1	0 1 0	1	0 0 1	Rm	1 1 1 1	Rd	1	0 1 1	Rm

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
REVSH<c> <Rd>,<Rm>
```

31 30 29 28	27 26 25 24 23	22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 1	1	1 1	(1) (1) (1) (1)	Rd	(1) (1) (1) (1)	1 0 1 1	Rm

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.137 REVSH

REVSH（符号付きハーフワードのバイト反転）は、32 ビットレジスタの下位 16 ビットハーフワードのバイト順序を反転して、結果を 32 ビットに符号拡張します。

エンコード T1 ARMv6*、ARMv7

```
REVSH<c> <Rd>,<Rm>
```

15 14 13 12	11 10 9 8	7 6	5 4 3	2 1 0
1 0 1 1	1 0 1 0	1 1	Rm	Rd

```
d = UInt(Rd);  m = UInt(Rm);
```

エンコード T2 ARMv6T2、ARMv7

```
REVSH<c>.W <Rd>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1	0 1 0	1	0 0 1	Rm	1 1 1 1	Rd	1	0 1 1	Rm

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
REVSH<c> <Rd>,<Rm>
```

31 30 29 28	27 26 25 24 23	22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 1	1	1 1	(1) (1) (1) (1)	Rd	(1) (1) (1) (1)	1 0 1 1	Rm

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
REVSH<c><q>  <Rd>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rm> The register that contains the operand. Its number must be encoded twice in encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:8>  = SignExtend(R[m]<7:0>, 24);
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
REVSH<c><q>  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rm> オペランドを含むレジスタ。エンコード T2 では、Rm の番号を 2 回エンコードする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:8>  = SignExtend(R[m]<7:0>, 24);
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.138 RFE

Return From Exception is a system instruction. For details see *RFE* on page B6-16.

A8.6.139 ROR (immediate)

Rotate Right (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. It can optionally update the condition flags based on the result.

Encoding T1 ARMv6T2, ARMv7

```
ROR{S}<c> <Rd>,<Rm>,#<imm>
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 0 1 0	S	1 1 1 1	(0)	imm3	Rd	imm2	1 1	Rm

```
if (imm3:imm2) == '00000' then SEE RRX;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('11', imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ROR{S}<c> <Rd>,<Rm>,#<imm>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5 4	3 2 1 0
cond	0 0	0	1 1 0 1	S	(0) (0) (0) (0)	Rd	imm5	1 1 0	Rm

```
if imm5 == '00000' then SEE RRX;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('11', imm5);
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.138 RFE

RFE（例外からの復帰）はシステム命令です。詳細については、*RFE* P. B6-16 を参照して下さい。

A8.6.139 ROR（イミディエート）

ROR（右ローテート（イミディエート））は、定数値で示されるビット数だけレジスタの内容をローテートします。ローテートによって右端からはみ出したビットは、左側の空きビットの位置に挿入されます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

```
ROR{S}<c> <Rd>,<Rm>,#<imm>
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 0 1 0	S	1 1 1 1	(0)	imm3	Rd	imm2	1 1	Rm

```
if (imm3:imm2) == '00000' then SEE RRX;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('11', imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ROR{S}<c> <Rd>,<Rm>,#<imm>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5 4	3 2 1 0
cond	0 0	0	1 1 0 1	S	(0) (0) (0) (0)	Rd	imm5	1 1 0	Rm

```
if imm5 == '00000' then SEE RRX;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('11', imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ROR{S}<c><q>  {<Rd>,} <Rm>, #<imm>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rm>	The first operand register.
<imm>	The shift amount, in the range 1 to 31. See *Shifts applied to a register* on page A8-10.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_ROR, shift_n, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ROR{S}<c><q>  {<Rd>,} <Rm>, #<imm>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rm>	最初のオペランドレジスタ
<imm>	1 ～ 31 の範囲のシフト量。詳細については、*レジスタに適用されるシフト* P. A8- 10 を参照して下さい。

動作


```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_ROR, shift_n, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```


例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.140 ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
RORS <Rdn>,<Rm>                    Outside IT block.
ROR<c> <Rdn>,<Rm>                  Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

Encoding T2 ARMv6T2, ARMv7

```
ROR{S}<c>.W <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	1	S	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ROR{S}<c> <Rd>,<Rn>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				Rm				0	1	1	1	Rn			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.140 ROR（レジスタ）

ROR（右ローテート（レジスタ））は、可変のビット数分だけレジスタの内容をローテートします。ローテートによって右端からはみ出したビットは、左側の空きビットの位置に挿入されます。ローテートするビット数は、レジスタの最下位バイトから読み出されます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
RORS <Rdn>,<Rm>                    IT ブロック外部
ROR<c> <Rdn>,<Rm>                  IT ブロック内部
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

エンコード T2 ARMv6T2、ARMv7

```
ROR{S}<c>.W <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	1	S	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
ROR{S}<c> <Rd>,<Rn>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd				Rm				0	1	1	1	Rn			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
ROR{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The register whose bottom byte contains the amount to rotate by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_ROR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
ROR{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	最下位バイトにローテートのビット数を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_ROR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.141 RRX

Rotate Right with Extend provides the value of the contentsof a register shifted right by one place, with the carry flag shifted into bit [31].

`RRX` can optionally update the condition flags based on the result. In that case, bit [0] is shifted into the carry flag.

Encoding T1 ARMv6T2, ARMv7

```
RRX{S}<c> <Rd>,<Rm>
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 0 1 0	S	1 1 1 1	(0)	0 0 0	Rd	0 0	1 1	Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
RRX{S}<c> <Rd>,<Rm>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
cond	0 0	0	1 1 0 1	S	(0) (0) (0) (0)	Rd	0 0 0 0	1 1	0	Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.141 RRX

RRX（拡張付き右ローテート）は、レジスタの内容を 1 ビット分右にシフトします。ビット [31] にはキャリーフラグがシフトされます。

`RRX` では、オプションとして、結果に基づいて条件フラグを更新することができます。この場合、ビット [0] がキャリーフラグにシフトされます。

エンコード T1 ARMv6T2、ARMv7

```
RRX{S}<c> <Rd>,<Rm>
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 0 1 0	S	1 1 1 1	(0)	0 0 0	Rd	0 0	1 1	Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
RRX{S}<c> <Rd>,<Rm>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
cond	0 0	0	1 1 0 1	S	(0) (0) (0) (0)	Rd	0 0 0 0	1 1	0	Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
RRX{S}<c><q>  {<Rd>,} <Rm>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rm>	The register that contains the operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_RRX, 1, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
RRX{S}<c><q>  {<Rd>,} <Rm>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rm>	オペランドを含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_RRX, 1, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.142 RSB (immediate)

Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
RSBS <Rd>,<Rn>,#0                    Outside IT block.
RSB<c> <Rd>,<Rn>,#0                  Inside IT block.
```

15 14 13 12 11 10	9	8 7 6 5	4 3 2 1 0	
0 1 0 0 0 0	1	0 0 1	Rn	Rd

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = Zeros(32); // immediate = #0
```

Encoding T2 ARMv6T2, ARMv7

```
RSB{S}<c>.W <Rd>,<Rn>,#<const>
```

15 14 13 12 11 10	9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	0	1 1 1 0	S	Rn	0	imm3	Rd	imm8

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
RSB{S}<c> <Rd>,<Rn>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 0 1 1	S	Rn	Rd	imm12

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.142 RSB（イミディエート）

RSB（逆減算（イミディエート））は、イミディエート値からレジスタの値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
RSBS <Rd>,<Rn>,#0                    IT ブロック外部
RSB<c> <Rd>,<Rn>,#0                  IT ブロック内部
```

15 14 13 12 11 10	9	8 7 6 5	4 3 2 1 0	
0 1 0 0 0 0	1	0 0 1	Rn	Rd

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = Zeros(32); // immediate = #0
```

エンコード T2 ARMv6T2、ARMv7

```
RSB{S}<c>.W <Rd>,<Rn>,#<const>
```

15 14 13 12 11 10	9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	0	1 1 1 0	S	Rn	0	imm3	Rd	imm8

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
RSB{S}<c> <Rd>,<Rn>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 0 1 1	S	Rn	Rd	imm12

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
RSB{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<const>	The immediate value to be added to the value obtained from <Rn>. The only permitted value for encoding T1 is 0. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values for encoding T2 or A1.

The pre-UAL syntax RSB<c>S is equivalent to RSBS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), imm32, '1');
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
RSB{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<const>	<Rn> から取得された値に加算されるイミディエート値。エンコード T1 で許容される値は 0 のみです。エンコード T2 または A1 で許容される値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

UAL 以前の構文の RSB<c>S は、RSBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), imm32, '1');
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.143 RSB (register)

Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv6T2, ARMv7

```
RSB{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	1 1 1 0	S	Rn	(0)	imm3	Rd	imm2	type	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
RSB{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
cond	0 0	0	0 0 1 1	S	Rn	Rd	imm5	type	0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.143 RSB（レジスタ）

RSB（逆減算（レジスタ））は、オプションとしてシフトされたレジスタの値からレジスタの値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

```
RSB{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	1 1 1 0	S	Rn	(0)	imm3	Rd	imm2	type	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
RSB{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
cond	0 0	0	0 0 1 1	S	Rn	Rd	imm5	type	0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
RSB{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The register that is optionally shifted and used as the second operand.
<shift>	The shift to apply to the value read from <Rm>. If omitted, no shift is applied. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

The pre-UAL syntax RSB<c>S is equivalent to RSBS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), shifted, '1');
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
RSB{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ
<shift>	<Rm> から読み出された値に適用されるシフト。省略した場合、シフトは適用されません。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

UAL 以前の構文の RSB<c>S は、RSBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), shifted, '1');
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.144 RSB (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
RSB{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0 0	0 0 1 1	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.144 RSB（レジスタによりシフトしたレジスタ）

RSB（逆減算（レジスタによりシフトしたレジスタ））は、レジスタによりシフトしたレジスタの値からレジスタの値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
RSB{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0 0	0 0 1 1	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
RSB{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The register that is shifted and used as the second operand.

<type> — The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR — Arithmetic shift right, encoded as type = 0b10
- LSL — Logical shift left, encoded as type = 0b00
- LSR — Logical shift right, encoded as type = 0b01
- ROR — Rotate right, encoded as type = 0b11.

<Rs> — The register whose bottom byte contains the amount to shift by.

The pre-UAL syntax RSB<c>S is equivalent to RSBS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), shifted, '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
RSB{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> — <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR — 算術右シフト。type = 0b10 としてエンコードされます。
- LSL — 論理左シフト。type = 0b00 としてエンコードされます。
- LSR — 論理右シフト。type = 0b01 としてエンコードされます。
- ROR — 右ローテート。type = 0b11 としてエンコードされます。

<Rs> — 最下位バイトにシフトのビット数を含むレジスタ

UAL 以前の構文の RSB<c>S は、RSBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), shifted, '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.145 RSC (immediate)

Reverse Subtract with Carry (immediate) subtracts a register value and the value of NOT (Carry flag) from an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
RSC{S}<c> <Rd>,<Rn>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 1 1 1	S	Rn	Rd	imm12

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.145 RSC（イミディエート）

RSC（キャリー付き逆減算（イミディエート））は、イミディエート値からレジスタの値とキャリーフラグの否定値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
RSC{S}<c> <Rd>,<Rn>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 1 1 1	S	Rn	Rd	imm12

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
RSC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<const>	The immediate value that the value obtained from <Rn> is to be subtracted from. See *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

The pre-UAL syntax RSC<c>S is equivalent to RSCS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), imm32, APSR.C);
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
RSC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<const>	<Rn> から取得された値が減算されるイミディエート値。値の範囲については、*ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

UAL 以前の構文の RSC<c>S は、RSCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), imm32, APSR.C);
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.146 RSC (register)

Reverse Subtract with Carry (register) subtracts a register value and the value of NOT (Carry flag) from an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
RSC{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 0	0	0 1 1 1	S	Rn	Rd	imm5	type	0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.146 RSC（レジスタ）

RSC（キャリー付き逆減算（レジスタ））は、オプションとしてシフトされたレジスタの値から、レジスタの値とキャリーフラグの否定値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
RSC{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 0	0	0 1 1 1	S	Rn	Rd	imm5	type	0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
RSC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The register that is optionally shifted and used as the second operand.
<shift>	The shift to apply to the value read from <Rm>. If omitted, no shift is applied. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

The pre-UAL syntax RSC<c>S is equivalent to RSCS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), shifted, APSR.C);
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
RSC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ
<shift>	<Rm> から読み出された値に適用されるシフト。省略した場合、シフトは適用されません。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

UAL 以前の構文の RSC<c>S は、RSCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), shifted, APSR.C);
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.147 RSC (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value and the value of NOT (Carry flag) from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
RSC{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 1 1 1	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.147 RSC（レジスタによりシフトしたレジスタ）

RSC（逆減算（レジスタによりシフトしたレジスタ））は、レジスタによりシフトしたレジスタの値から、レジスタの値とキャリーフラグの否定値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
RSC{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 1 1 1	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
RSC{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The register that is shifted and used as the second operand.

<type> — The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR — Arithmetic shift right, encoded as type = 0b10
- LSL — Logical shift left, encoded as type = 0b00
- LSR — Logical shift right, encoded as type = 0b01
- ROR — Rotate right, encoded as type = 0b11.

<Rs> — The register whose bottom byte contains the amount to shift by.

The pre-UAL syntax RSC<c>S is equivalent to RSCS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), shifted, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
RSC{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> — <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR — 算術右シフト。type = 0b10 としてエンコードされます。
- LSL — 論理左シフト。type = 0b00 としてエンコードされます。
- LSR — 論理右シフト。type = 0b01 としてエンコードされます。
- ROR — 右ローテート。type = 0b11 としてエンコードされます。

<Rs> — 最下位バイトにシフトのビット数を含むレジスタ

UAL 以前の構文の RSC<c>S は、RSCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), shifted, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.148 SADD16

Signed Add 16 performs two 16-bit signed integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

Encoding T1 ARMv6T2, ARMv7

```
SADD16<c> <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SADD16<c> <Rd>,<Rn>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	0	0	0	1	Rn				Rd				(1)	(1)	(1)	(1)	0	0	0	1	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.148 SADD16

SADD16（符号付き加算 16）は、2 つの符号付き 16 ビット整数加算を実行して、結果をデスティネーションレジスタに書き込みます。加算の結果に従って APSR.GE ビットをセットします。

エンコード T1 ARMv6T2、ARMv7

```
SADD16<c> <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SADD16<c> <Rd>,<Rn>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	0	0	0	1	Rn				Rd				(1)	(1)	(1)	(1)	0	0	0	1	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SADD16<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<15:0>;
    R[d]<31:16> = sum2<15:0>;
    APSR.GE<1:0> = if sum1 >= 0 then '11' else '00';
    APSR.GE<3:2> = if sum2 >= 0 then '11' else '00';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SADD16<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<15:0>;
    R[d]<31:16> = sum2<15:0>;
    APSR.GE<1:0> = if sum1 >= 0 then '11' else '00';
    APSR.GE<3:2> = if sum2 >= 0 then '11' else '00';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.149 SADD8

Signed Add 8 performs four 8-bit signed integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

Encoding T1 ARMv6T2, ARMv7

```
SADD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SADD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 0 0 1	Rn	Rd	(1) (1) (1) (1)	1	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.149 SADD8

SADD8（符号付き加算 8）は、4 つの符号付き 8 ビット整数加算を実行して、結果をデスティネーションレジスタに書き込みます。加算の結果に従って APSR.GE ビットをセットします。

エンコード T1 ARMv6T2、ARMv7

```
SADD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SADD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 0 0 1	Rn	Rd	(1) (1) (1) (1)	1	0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SADD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0>   = sum1<7:0>;
    R[d]<15:8>  = sum2<7:0>;
    R[d]<23:16> = sum3<7:0>;
    R[d]<31:24> = sum4<7:0>;
    APSR.GE<0>  = if sum1 >= 0 then '1' else '0';
    APSR.GE<1>  = if sum2 >= 0 then '1' else '0';
    APSR.GE<2>  = if sum3 >= 0 then '1' else '0';
    APSR.GE<3>  = if sum4 >= 0 then '1' else '0';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SADD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0>   = sum1<7:0>;
    R[d]<15:8>  = sum2<7:0>;
    R[d]<23:16> = sum3<7:0>;
    R[d]<31:24> = sum4<7:0>;
    APSR.GE<0>  = if sum1 >= 0 then '1' else '0';
    APSR.GE<1>  = if sum2 >= 0 then '1' else '0';
    APSR.GE<2>  = if sum3 >= 0 then '1' else '0';
    APSR.GE<3>  = if sum4 >= 0 then '1' else '0';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.150 SASX

Signed Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

Encoding T1 ARMv6T2, ARMv7

```
SASX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SASX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 0 1	Rn	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.150 SASX

SASX（交換付き符号付き加算および減算）は、2 番目のオペランドの 2 つのハーフワードを交換し、1 つの 16 ビット整数加算と 1 つの 16 ビット減算を実行して、結果をデスティネーションレジスタに書き込みます。結果に従って APSR.GE ビットをセットします。

エンコード T1 ARMv6T2、ARMv7

```
SASX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SASX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 0 1	Rn	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SASX<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<Rm> The second operand register.

The pre-UAL syntax SADDSUBX<c> is equivalent to SASX<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>  = diff<15:0>;
    R[d]<31:16> = sum<15:0>;
    APSR.GE<1:0> = if diff >= 0 then '11' else '00';
    APSR.GE<3:2> = if sum  >= 0 then '11' else '00';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SASX<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<Rm> 2 番目のオペランドレジスタ

UAL 以前の構文の SADDSUBX<c> は、SASX<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>  = diff<15:0>;
    R[d]<31:16> = sum<15:0>;
    APSR.GE<1:0> = if diff >= 0 then '11' else '00';
    APSR.GE<3:2> = if sum  >= 0 then '11' else '00';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.151 SBC (immediate)

Subtract with Carry (immediate) subtracts an immediate value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv6T2, ARMv7

```
SBC{S}<c> <Rd>,<Rn>,#<const>
```

15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	0	1 0 1 1	S	Rn	0	imm3	Rd	imm8

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
SBC{S}<c> <Rd>,<Rn>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 1 1 0	S	Rn	Rd	imm12

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.151 SBC（イミディエート）

SBC（キャリー付き減算（イミディエート））は、レジスタの値からイミディエート値とキャリーフラグの否定値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

```
SBC{S}<c> <Rd>,<Rn>,#<const>
```

15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	0	1 0 1 1	S	Rn	0	imm3	Rd	imm8

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
SBC{S}<c> <Rd>,<Rn>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 1 1 0	S	Rn	Rd	imm12

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SBC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<const>	The immediate value to be subtracted from the value obtained from <Rn>. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

The pre-UAL syntax SBC<c>S is equivalent to SBCS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SBC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<const>	<Rn> から取得された値から減算されるイミディエート値。値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

UAL 以前の構文の SBC<c>S は、SBCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.152 SBC (register)

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
SBCS <Rdn>,<Rm>          Outside IT block.
SBC<c> <Rdn>,<Rm>        Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
SBC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	1	1	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
SBC{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	1	1	0	S	Rn				Rd				imm5					type		0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.152 SBC（レジスタ）

SBC（キャリー付き減算（レジスタ））は、レジスタの値から、オプションとしてシフトされたレジスタの値とキャリーフラグの否定値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
SBCS <Rdn>,<Rm>          IT ブロック外部
SBC<c> <Rdn>,<Rm>        IT ブロック内部
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
SBC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	1	1	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
SBC{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	0	1	1	0	S	Rn				Rd				imm5					type		0	Rm			

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SBC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The register that is optionally shifted and used as the second operand.
<shift>	The shift to apply to the value read from <Rm>. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

The pre-UAL syntax SBC<c>S is equivalent to SBCS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SBC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ
<shift>	<Rm> から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

UAL 以前の構文の SBC<c>S は、SBCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.153 SBC (register-shifted register)

Subtract with Carry (register-shifted register) subtracts a register-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
SBC{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 1 1 0	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.153 SBC（レジスタによりシフトしたレジスタ）

SBC（キャリー付き減算（レジスタによりシフトしたレジスタ））は、レジスタの値から、レジスタによりシフトしたレジスタの値とキャリーフラグの否定値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
SBC{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 1 1 0	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SBC{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The register that is shifted and used as the second operand.

<type> — The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR — Arithmetic shift right, encoded as type = 0b10
- LSL — Logical shift left, encoded as type = 0b00
- LSR — Logical shift right, encoded as type = 0b01
- ROR — Rotate right, encoded as type = 0b11.

<Rs> — The register whose bottom byte contains the amount to shift by.

The pre-UAL syntax SBC<c>S is equivalent to SBCS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SBC{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> — <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR — 算術右シフト。type = 0b10 としてエンコードされます。
- LSL — 論理左シフト。type = 0b00 としてエンコードされます。
- LSR — 論理右シフト。type = 0b01 としてエンコードされます。
- ROR — 右ローテート。type = 0b11 としてエンコードされます。

<Rs> — 最下位バイトにシフトのビット数を含むレジスタ

UAL 以前の構文の SBC<c>S は、SBCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.154 SBFX

Signed Bit Field Extract extracts any number of adjacent bits at any position from a register, sign-extends them to 32 bits, and writes the result to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
SBFX<c> <Rd>,<Rn>,#<lsb>,#<width>
```

15 14 13 12 11	10	9 8	7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5	4 3 2 1 0
1 1 1 1 0	(0)	1 1	0 1 0	0	Rn	0	imm3	Rd	imm2	(0)	widthm1

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv6T2, ARMv7

```
SBFX<c> <Rd>,<Rn>,#<lsb>,#<width>
```

31 30 29 28	27 26 25 24 23	22	21	20 19 18 17 16	15 14 13 12	11 10 9 8 7	6 5 4	3 2 1 0
cond	0 1 1 1 1	0	1	widthm1	Rd	lsb	1 0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(lsb);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.154 SBFX

SBFX（符号付きビットフィールドの抽出）は、1 つのレジスタの任意の位置にある、任意の数の隣接するビットを抽出して 32 ビットに符号拡張し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
SBFX<c> <Rd>,<Rn>,#<lsb>,#<width>
```

15 14 13 12 11	10	9 8	7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5	4 3 2 1 0
1 1 1 1 0	(0)	1 1	0 1 0	0	Rn	0	imm3	Rd	imm2	(0)	widthm1

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv6T2、ARMv7

```
SBFX<c> <Rd>,<Rn>,#<lsb>,#<width>
```

31 30 29 28	27 26 25 24 23	22	21	20 19 18 17 16	15 14 13 12	11 10 9 8 7	6 5 4	3 2 1 0
cond	0 1 1 1 1	0	1	widthm1	Rd	lsb	1 0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(lsb);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SBFX<c><q>  <Rd>, <Rn>, #<lsb>, #<width>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<lsb> is the bit number of the least significant bit in the bitfield, in the range 0-31. This determines the required value of lsbit.

<width> is the width of the bitfield, in the range 1 to 32-<lsb>. The required value of widthminus1 is <width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = SignExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SBFX<c><q>  <Rd>, <Rn>, #<lsb>, #<width>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<lsb> ビットフィールド内の最下位ビットのビット番号で、範囲は 0 ～ 31 です。これによって、lsbit に必要な値が決定されます。

<width> ビットフィールドの幅で、範囲は 1 ～ 32 – <lsb> です。widthminus1 に必要な値は <width> - 1 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = SignExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.155 SDIV

Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. The condition code flags are not affected.

Encoding T1 ARMv7-R

```
SDIV<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 1 0 0	1	Rn	(1) (1) (1) (1)	Rd	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.155 SDIV

SDIV（符号付き除算）は、32 ビットの符号付き整数レジスタの値を、32 ビットの符号付き整数レジスタの値で除算して、結果をデスティネーションレジスタに書き込みます。条件コードフラグは影響を受けません。

エンコード T1 ARMv7-R

```
SDIV<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 1 0 0	1	Rn	(1) (1) (1) (1)	Rd	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SDIV<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The register that contains the dividend.
`<Rm>`	The register that contains the divisor.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if SInt(R[m]) == 0 then
        if IntegerZeroDivideTrappingEnabled() then
            GenerateIntegerZeroDivide();
        else
            result = 0;
    else
        result = RoundTowardsZero(SInt(R[n]) / SInt(R[m]));
    R[d] = result<31:0>;
```

Exceptions

Undefined Instruction.

Overflow

If the signed integer division `0x80000000 / 0xFFFFFFFF` is performed, the pseudocode produces the intermediate integer result $+2^{31}$, that overflows the 32-bit signed integer range. No indication of this overflow case is produced, and the 32-bit result written to `R[d]` must be the bottom 32 bits of the binary representation of $+2^{31}$. So the result of the division is `0x80000000`.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SDIV<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	被除数を含むレジスタ
`<Rm>`	除数を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if SInt(R[m]) == 0 then
        if IntegerZeroDivideTrappingEnabled() then
            GenerateIntegerZeroDivide();
        else
            result = 0;
    else
        result = RoundTowardsZero(SInt(R[n]) / SInt(R[m]));
    R[d] = result<31:0>;
```

例外

未定義命令

オーバフロー

符号付き整数除算の「`0x80000000/0xFFFFFFFF`」が実行された場合は、擬似コードによって、32 ビット符号付き整数の範囲を超える、$+2^{31}$ の中間整数結果が生成されます。これによるオーバフローは発生しないため、`R[d]` に書き込む 32 ビットの結果を $+2^{31}$ のバイナリ表現の下位 32 ビットにする必要があります。そのため、除算の結果は `0x80000000` になります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.156 SEL

Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the GE flags.

Encoding T1 ARMv6T2, ARMv7

```
SEL<c> <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	0	Rn				1	1	1	1	Rd				1	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SEL<c> <Rd>,<Rn>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	0	0	Rn				Rd				(1)	(1)	(1)	(1)	1	0	1	1	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.156 SEL

SEL（バイト選択）は、GE フラグの値に従って、最初のオペランドまたは 2 番目のオペランドから結果の各バイトを選択します。

エンコード T1 ARMv6T2、ARMv7

```
SEL<c> <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	0	Rn				1	1	1	1	Rd				1	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SEL<c> <Rd>,<Rn>,<Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	0	0	Rn				Rd				(1)	(1)	(1)	(1)	1	0	1	1	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SEL<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<7:0>   = if APSR.GE<0> == '1' then R[n]<7:0>   else R[m]<7:0>;
    R[d]<15:8>  = if APSR.GE<1> == '1' then R[n]<15:8>  else R[m]<15:8>;
    R[d]<23:16> = if APSR.GE<2> == '1' then R[n]<23:16> else R[m]<23:16>;
    R[d]<31:24> = if APSR.GE<3> == '1' then R[n]<31:24> else R[m]<31:24>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SEL<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<7:0>   = if APSR.GE<0> == '1' then R[n]<7:0>   else R[m]<7:0>;
    R[d]<15:8>  = if APSR.GE<1> == '1' then R[n]<15:8>  else R[m]<15:8>;
    R[d]<23:16> = if APSR.GE<2> == '1' then R[n]<23:16> else R[m]<23:16>;
    R[d]<31:24> = if APSR.GE<3> == '1' then R[n]<31:24> else R[m]<31:24>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.157 SETEND

Set Endianness writes a new value to ENDIANSTATE.

Encoding T1 ARMv6*, ARMv7

```
SETEND <endian_specifier>
```

Not permitted in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	0	(1)	E	(0)	(0)	(0)

```
set_bigend = (E == '1');
if InITBlock() then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SETEND <endian_specifier>
```

Cannot be conditional

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)	(0)	(0)	E	(0)	0	0	0	0	(0)	(0)	(0)	(0)

```
set_bigend = (E == '1');
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.157 SETEND

SETEND（エンディアン形式の設定）は、新しい値を ENDIANSTATE に書き込みます。

エンコード T1 ARMv6*、ARMv7

```
SETEND <endian_specifier>
```

IT ブロック内部では許可されない

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	0	(1)	E	(0)	(0)	(0)

```
set_bigend = (E == '1');
if InITBlock() then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SETEND <endian_specifier>
```

条件付きにすることはできない

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)	(0)	(0)	E	(0)	0	0	0	0	(0)	(0)	(0)	(0)

```
set_bigend = (E == '1');
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SETEND<q>  <endian_specifier>
```

where:

<q>
See *Standard assembler syntax fields* on page A8-7. A SETEND instruction must be unconditional.

<endian_specifier>
Is one of:

BE
Sets the E bit in the instruction. This sets ENDIANSTATE.

LE
Clears the E bit in the instruction. This clears ENDIANSTATE.

Operation

```
EncodingSpecificOperations();
ENDIANSTATE = if set_bigend then '1' else '0';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SETEND<q>  <endian_specifier>
```

各項目の説明については以下を参照して下さい。

<q>
標準アセンブラ構文のフィールド P. A8-7 を参照して下さい。SETEND は常に無条件命令です。

<endian_specifier>
次のいずれかを指定します。

BE
命令の E ビットをセットします。これにより ENDIANSTATE がセットされます。

LE
命令の E ビットをクリアします。これにより ENDIANSTATE がクリアされます。

動作

```
EncodingSpecificOperations();
ENDIANSTATE = if set_bigend then '1' else '0';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.158 SEV

Send Event is a hint instruction. It causes an event to be signaled to all processors in the multiprocessor system.

Encoding T1 ARMv7 (executes as NOP in ARMv6T2)

```
SEV<c>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0

```
// No additional decoding required
```

Encoding T2 ARMv7 (executes as NOP in ARMv6T2)

```
SEV<c>.W
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	1	0	0

```
// No additional decoding required
```

Encoding A1 ARMv6K, ARMv7 (executes as NOP in ARMv6T2)

```
SEV<c>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	1	0	0

```
// No additional decoding required
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.158 SEV

SEV（イベント送信）はヒント命令です。マルチプロセッサシステム内部のすべてのプロセッサに伝達されるイベントを発生させます。

エンコード T1 ARMv7（ARMv6T2 では NOP として動作する）

```
SEV<c>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0

```
// No additional decoding required
```

エンコード T2 ARMv7（ARMv6T2 では NOP として動作する）

```
SEV<c>.W
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	1	0	0

```
// No additional decoding required
```

エンコード A1 ARMv6K、ARMv7（ARMv6T2 では NOP として動作する）

```
SEV<c>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	1	0	0

```
// No additional decoding required
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SEV<c><q>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SEV<c><q>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.159 SHADD16

Signed Halving Add 16 performs two signed 16-bit integer additions, halves the results, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
SHADD16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SHADD16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 1	Rn	Rd	(1) (1) (1) (1)	0 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.159 SHADD16

SHADD16（符号付き加算半減 16）は、2 つの符号付き 16 ビット整数加算を実行し、結果を半減し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
SHADD16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SHADD16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 1	Rn	Rd	(1) (1) (1) (1)	0 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SHADD16<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<16:1>;
    R[d]<31:16> = sum2<16:1>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SHADD16<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<16:1>;
    R[d]<31:16> = sum2<16:1>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.160 SHADD8

Signed Halving Add 8 performs four signed 8-bit integer additions, halves the results, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
SHADD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SHADD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 1	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.160 SHADD8

SHADD8（符号付き加算半減 8）は、4 つの 8 ビット符号付き整数加算を実行し、結果を半減し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
SHADD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SHADD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 1	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SHADD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0>   = sum1<8:1>;
    R[d]<15:8>  = sum2<8:1>;
    R[d]<23:16> = sum3<8:1>;
    R[d]<31:24> = sum4<8:1>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SHADD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0>   = sum1<8:1>;
    R[d]<15:8>  = sum2<8:1>;
    R[d]<23:16> = sum3<8:1>;
    R[d]<31:24> = sum4<8:1>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.161 SHASX

Signed Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer addition and one signed 16-bit subtraction, halves the results, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
SHASX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SHASX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 1	Rn	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.161 SHASX

SHASX（交換付き符号付き加算および減算半減）は、2 番目のオペランドの 2 つのハーフワードを交換し、1 つの符号付き 16 ビット整数加算と 1 つの符号付き 16 ビット減算を実行し、結果を半減し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
SHASX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SHASX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 1	Rn	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SHASX<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<Rm> The second operand register.

The pre-UAL syntax SHADDSUBX<c> is equivalent to SHASX<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>  = diff<16:1>;
    R[d]<31:16> = sum<16:1>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SHASX<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<Rm> 2 番目のオペランドレジスタ

UAL 以前の構文の SHADDSUBX<c> は、SHASX<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>  = diff<16:1>;
    R[d]<31:16> = sum<16:1>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.162 SHSAX

Signed Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer subtraction and one signed 16-bit addition, halves the results, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
SHSAX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 1 0	Rn	1 1 1 1	Rd	0	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SHSAX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 1	Rn	Rd	(1) (1) (1) (1)	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.162 SHSAX

SHSAX（交換付き符号付き減算および加算半減）は、2 番目のオペランドの 2 つのハーフワードを交換し、1 つの符号付き 16 ビット整数減算と 1 つの符号付き 16 ビット加算を実行し、結果を半減し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
SHSAX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 1 0	Rn	1 1 1 1	Rd	0	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SHSAX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 1	Rn	Rd	(1) (1) (1) (1)	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SHSAX<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<Rm> The second operand register.

The pre-UAL syntax SHSUBADDX<c> is equivalent to SHSAX<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0>  = sum<16:1>;
    R[d]<31:16> = diff<16:1>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SHSAX<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<Rm> 2 番目のオペランドレジスタ

UAL 以前の構文の SHSUBADDX<c> は、SHSAX<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0>  = sum<16:1>;
    R[d]<31:16> = diff<16:1>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.163 SHSUB16

Signed Halving Subtract 16 performs two signed 16-bit integer subtractions, halves the results, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
SHSUB16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SHSUB16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 1	Rn	Rd	(1) (1) (1) (1)	0	1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.163 SHSUB16

SHSUB16（符号付き減算半減 16）は、2 つの符号付き 16 ビット整数減算を実行し、結果を半減し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
SHSUB16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SHSUB16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 1	Rn	Rd	(1) (1) (1) (1)	0	1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SHSUB16<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<16:1>;
    R[d]<31:16> = diff2<16:1>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SHSUB16<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<16:1>;
    R[d]<31:16> = diff2<16:1>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.164 SHSUB8

Signed Halving Subtract 8 performs four signed 8-bit integer subtractions, halves the results, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
SHSUB8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 0	Rn	1 1 1 1	Rd	0	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SHSUB8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 1	Rn	Rd	(1) (1) (1) (1)	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.164 SHSUB8

SHSUB8（符号付き減算半減 8）は、4 つの符号付き 8 ビット整数減算を実行し、結果を半減し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
SHSUB8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 0	Rn	1 1 1 1	Rd	0	0 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SHSUB8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 0 1 1	Rn	Rd	(1) (1) (1) (1)	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SHSUB8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0>   = diff1<8:1>;
    R[d]<15:8>  = diff2<8:1>;
    R[d]<23:16> = diff3<8:1>;
    R[d]<31:24> = diff4<8:1>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SHSUB8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0>   = diff1<8:1>;
    R[d]<15:8>  = diff2<8:1>;
    R[d]<23:16> = diff3<8:1>;
    R[d]<31:24> = diff4<8:1>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.165 SMC (previously SMI)

Secure Monitor Call is a system instruction. For details see *SMC (previously SMI)* on page B6-18.

A8.6.166 SMLABB, SMLABT, SMLATB, SMLATT

Signed Multiply Accumulate (halfwords) performs a signed multiply-accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets the Q flag in the APSR. It is not possible for overflow to occur during the multiplication.

Encoding T1 ARMv6T2, ARMv7

```
SMLA<x><y><c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 0 1	Rn	Ra	Rd	0 0	N	M	Rm

```
if Ra == '1111' then SEE SMULBB, SMULBT, SMULTB, SMULTT;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
SMLA<x><y><c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
cond	0 0 0 1	0 0 0 0	Rd	Ra	Rm	1	M	N	0	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.165 SMC（以前の SMI）

SMC（セキュア モニタコール）はシステム命令です。詳細については、*SMC（以前の SMI）* P. B6- 18 を参照して下さい。

A8.6.166 SMLABB、SMLABT、SMLATB、SMLATT

符号付き積和（ハーフワード）は、符号付き積和演算を実行します。それぞれのソースレジスタの下位半分または上位半分から取得された、2 つの符号付き 16 ビット数値に対して乗算が実行されます。これらのソースレジスタの他の半分は無視されます。32 ビットの積が 32 ビットの累積値に加算され、結果がデスティネーションレジスタに書き込まれます。

累積値の加算中にオーバフローが発生すると、APSR の Q フラグがセットされます。乗算中にオーバフローが発生することはありません。

エンコード T1 ARMv6T2、ARMv7

```
SMLA<x><y><c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 0 1	Rn	Ra	Rd	0 0	N	M	Rm

```
if Ra == '1111' then SEE SMULBB, SMULBT, SMULTB, SMULTT;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
SMLA<x><y><c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
cond	0 0 0 1	0 0 0 0	Rd	Ra	Rm	1	M	N	0	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMLA<x><y><c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

where:

<x> Specifies which half of the source register <Rn> is used as the first multiply operand. If <x> is B, then the bottom half (bits [15:0]) of <Rn> is used. If <x> is T, then the top half (bits [31:16]) of <Rn> is used.

<y> Specifies which half of the source register <Rm> is used as the second multiply operand. If <y> is B, then the bottom half (bits [15:0]) of <Rm> is used. If <y> is T, then the top half (bits [31:16]) of <Rm> is used.

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The source register whose bottom or top half (selected by <x>) is the first multiply operand.

<Rm> The source register whose bottom or top half (selected by <y>) is the second multiply operand.

<Ra> The register that contains the accumulate value.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMLA<x><y><c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

各項目の説明については以下を参照して下さい。

<x> ソースレジスタ <Rn> のどの半分を最初の乗算オペランドとして使用するかを指定します。<x> が B の場合、<Rn> の下位半分（ビット [15:0]）が使用されます。<x> が T の場合、<Rn> の上位半分（ビット [31:16]）が使用されます。

<y> ソースレジスタ <Rm> のどの半分を 2 番目の乗算オペランドとして使用するかを指定します。<y> が B の場合、<Rm> の下位半分（ビット [15:0]）が使用されます。<y> が T の場合、<Rm> の上位半分（ビット [31:16]）が使用されます。

<c><q> *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 下位半分または上位半分（<x> によって選択される）が最初の乗算オペランドである、ソースレジスタ

<Rm> 下位半分または上位半分（<y> によって選択される）が 2 番目の乗算オペランドである、ソースレジスタ

<Ra> 累積値を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.167 SMLAD

Signed Multiply Accumulate Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top × bottom and bottom × top multiplication.

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications.

Encoding T1 ARMv6T2, ARMv7

```
SMLAD{X}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 1 0	Rn	Ra	Rd	0 0 0	M	Rm

```
if Ra == '1111' then SEE SMUAD;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
m_swap = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SMLAD{X}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1	0 0 0 0	Rd	Ra	Rm	0 0	M	1	Rn

```
if Ra == '1111' then SEE SMUAD;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.167 SMLAD

SMLAD（符号付きデュアル積和演算）は、2 つの符号付き 16 × 16 ビット乗算を実行します。積を 32 ビット累積オペランドに加算します。

オプションとして、この算術演算を実行する前に 2 番目のオペランドのハーフワード同士を交換することができます。これにより、上位 × 下位、および下位 × 上位の乗算が生成されます。

累積演算中にオーバフローが発生すると、Q フラグがセットされます。乗算中にオーバフローが発生することはありません。

エンコード T1 ARMv6T2、ARMv7

```
SMLAD{X}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 1 0	Rn	Ra	Rd	0 0 0	M	Rm

```
if Ra == '1111' then SEE SMUAD;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
m_swap = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SMLAD{X}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1	0 0 0 0	Rd	Ra	Rm	0 0	M	1	Rn

```
if Ra == '1111' then SEE SMUAD;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMLAD{X}<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

where:

X — If X is present (encoded as M == 1), the multiplications are bottom × top and top × bottom.
If the X is omitted (encoded as M == 0), the multiplications are bottom × bottom and top × top.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The second operand register.

<Ra> — The register that contains the accumulate value.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
SMLAD{X}<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

各項目の説明については以下を参照して下さい。

X — X を指定した場合（M == 1 としてエンコード）、乗算は下位 × 上位、および上位 × 下位となります。
X を省略した場合（M == 0 としてエンコード）、乗算は下位 × 下位、および上位 × 上位となります。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — 2 番目のオペランドレジスタ

<Ra> — 累積値を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.168 SMLAL

Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In ARM code, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many processor implementations.

Encoding T1 ARMv6T2, ARMv7

```
SMLAL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6	5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 1	1	0 0	Rn	RdLo	RdHi	0 0 0 0	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
SMLAL{S}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 1 1 1	S	RdHi	RdLo	Rm	1 0 0 1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
if ArchVersion() < 6 && (dHi == n || dLo == n) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.168 SMLAL

SMLAL（符号付きロング積和）は、2 つの符号付き 32 ビット値を乗算して 64 ビット値を生成し、別の 64 ビット値と加算します。

ARM コードでは、オプションとして、結果に基づいて条件フラグを更新することができます。このオプションを使用すると、多くのプロセッサ実装で性能が低下します。

エンコード T1 ARMv6T2、ARMv7

```
SMLAL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6	5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 1	1	0 0	Rn	RdLo	RdHi	0 0 0 0	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
SMLAL{S}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 1 1 1	S	RdHi	RdLo	Rm	1 0 0 1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
if ArchVersion() < 6 && (dHi == n || dLo == n) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMLAL{S}<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
S can be specified only for the ARM instruction set.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<RdLo> — Supplies the lower 32 bits of the accumulate value, and is the destination register for the lower 32 bits of the result.

<RdHi> — Supplies the upper 32 bits of the accumulate value, and is the destination register for the upper 32 bits of the result.

<Rn> — The first operand register.

<Rm> — The second operand register.

The pre-UAL syntax SMLAL<c>S is equivalent to SMLALS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        APSR.N = result<63>;
        APSR.Z = IsZeroBit(result<63:0>);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
            APSR.V = bit UNKNOWN;
        // else APSR.C, APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMLAL{S}<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
S は、ARM 命令セットでのみ指定できます。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<RdLo> — 加算される値の下位 32 ビットで、同時に結果の下位 32 ビットのデスティネーションレジスタとなります。

<RdHi> — 加算される値の上位 32 ビットで、同時に結果の上位 32 ビットのデスティネーションレジスタとなります。

<Rn> — 最初のオペランドレジスタ

<Rm> — 2 番目のオペランドレジスタ

UAL 以前の構文の SMLAL<c>S は、SMLALS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        APSR.N = result<63>;
        APSR.Z = IsZeroBit(result<63:0>);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
            APSR.V = bit UNKNOWN;
        // else APSR.C, APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.169 SMLALBB, SMLALBT, SMLALTB, SMLALTT

Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and accumulated with a 64-bit accumulate value.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

Encoding T1 ARMv6T2, ARMv7

```
SMLAL<x><y><c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
1 1 1 1 1	0 1 1 1	1 0 0	Rn	RdLo	RdHi	1 0	N	M	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
SMLAL<x><y><c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
cond	0 0 0 1 0 1 0 0	RdHi	RdLo	Rm	1	M	N	0	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.169 SMLALBB、SMLALBT、SMLALTB、SMLALTT

SMLALBB、SMLALBT、SMLALTB、SMLALTT（符号付きロング積和演算（ハーフワード））は、2 つの符号付き 16 ビット値を乗算して 32 ビット値を生成し、別の 64 ビット値と加算します。それぞれのソースレジスタの下位半分または上位半分から取得された、2 つの符号付き 16 ビット数値に対して乗算が実行されます。これらのソースレジスタの他の半分は無視されます。32 ビットの積が符号拡張され、64 ビットの累積値と加算されます。

この命令の実行中にオーバフローが発生することがありますが、64 ビット加算の結果としてのみ発生します。このオーバフローは、発生しても検出されません。その代わりに、結果がモジュロ 2^{64} にラップアラウンドされます。

エンコード T1 ARMv6T2、ARMv7

```
SMLAL<x><y><c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
1 1 1 1 1	0 1 1 1	1 0 0	Rn	RdLo	RdHi	1 0	N	M	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
SMLAL<x><y><c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
cond	0 0 0 1 0 1 0 0	RdHi	RdLo	Rm	1	M	N	0	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMLAL<x><y><c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

where:

<x> Specifies which half of the source register <Rn> is used as the first multiply operand. If <x> is B, then the bottom half (bits [15:0]) of <Rn> is used. If <x> is T, then the top half (bits [31:16]) of <Rn> is used.

<y> Specifies which half of the source register <Rm> is used as the second multiply operand. If <y> is B, then the bottom half (bits [15:0]) of <Rm> is used. If <y> is T, then the top half (bits [31:16]) of <Rm> is used.

<c><q> See *Standard assembler syntax fields* on page A8-7.

<RdLo> Supplies the lower 32 bits of the accumulate value, and is the destination register for the lower 32 bits of the result.

<RdHi> Supplies the upper 32 bits of the accumulate value, and is the destination register for the upper 32 bits of the result.

<Rn> The source register whose bottom or top half (selected by <x>) is the first multiply operand.

<Rm> The source register whose bottom or top half (selected by <y>) is the second multiply operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMLAL<x><y><c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<x> ソースレジスタ <Rn> のどの半分を最初の乗算オペランドとして使用するかを指定します。<x> が B の場合、<Rn> の下位半分（ビット [15:0]）が使用されます。<x> が T の場合、<Rn> の上位半分（ビット [31:16]）が使用されます。

<y> ソースレジスタ <Rm> のどの半分を 2 番目の乗算オペランドとして使用するかを指定します。<y> が B の場合、<Rm> の下位半分（ビット [15:0]）が使用されます。<y> が T の場合、<Rm> の上位半分（ビット [31:16]）が使用されます。

<c><q> *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

<RdLo> 加算される値の下位 32 ビットで、同時に結果の下位 32 ビットのデスティネーションレジスタとなります。

<RdHi> 加算される値の上位 32 ビットで、同時に結果の上位 32 ビットのデスティネーションレジスタとなります。

<Rn> 下位半分または上位半分（<x> によって選択される）が最初の乗算オペランドである、ソースレジスタ

<Rm> 下位半分または上位半分（<y> によって選択される）が 2 番目の乗算オペランドである、ソースレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.170 SMLALD

Signed Multiply Accumulate Long Dual performs two signed 16 × 16-bit multiplications. It adds the products to a 64-bit accumulate operand.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top × bottom and bottom × top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

Encoding T1 ARMv6T2, ARMv7

```
SMLALD{X}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
1 1 1 1 1	0 1 1 1	1 0 0	Rn	RdLo	RdHi	1 1	0	M	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SMLALD{X}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1 0 1 0 0	RdHi	RdLo	Rm	0 0	M	1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.170 SMLALD

SMLALD（符号付きデュアルロング積和演算）は、2 つの符号付き 16 × 16 ビット乗算を実行します。積を 64 ビット累積オペランドに加算します。

オプションとして、この算術演算を実行する前に 2 番目のオペランドのハーフワード同士を交換することができます。これにより、上位 × 下位、および下位 × 上位の乗算が生成されます。

この命令の実行中にオーバフローが発生することがありますが、64 ビット加算の結果としてのみ発生します。このオーバフローは、発生しても検出されません。その代わりに、結果がモジュロ 2^{64} にラップアラウンドされます。

エンコード T1 ARMv6T2、ARMv7

```
SMLALD{X}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
1 1 1 1 1	0 1 1 1	1 0 0	Rn	RdLo	RdHi	1 1	0	M	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SMLALD{X}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1 0 1 0 0	RdHi	RdLo	Rm	0 0	M	1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMLALD{X}<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

where:

X — If X is present, the multiplications are bottom × top and top × bottom.
If the X is omitted, the multiplications are bottom × bottom and top × top.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<RdLo> — Supplies the lower 32 bits of the accumulate value, and is the destination register for the lower 32 bits of the result.

<RdHi> — Supplies the upper 32 bits of the accumulate value, and is the destination register for the upper 32 bits of the result.

<Rn> — The first operand register.

<Rm> — The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMLALD{X}<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

X — X を指定した場合、乗算は下位 × 上位、および上位 × 下位となります。
X を省略した場合、乗算は下位 × 下位、および上位 × 上位となります。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<RdLo> — 加算される値の下位 32 ビットで、同時に結果の下位 32 ビットのデスティネーションレジスタとなります。

<RdHi> — 加算される値の上位 32 ビットで、同時に結果の上位 32 ビットのデスティネーションレジスタとなります。

<Rn> — 最初のオペランドレジスタ

<Rm> — 2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.171 SMLAWB, SMLAWT

Signed Multiply Accumulate (word by halfword) performs a signed multiply-accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored.

If overflow occurs during the addition of the accumulate value, the instruction sets the Q flag in the APSR. No overflow can occur during the multiplication.

Encoding T1 ARMv6T2, ARMv7

```
SMLAW<y><c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 1 1	Rn	Ra	Rd	0 0 0	M	Rm

```
if Ra == '1111' then SEE SMULWB, SMULWT;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
SMLAW<y><c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
cond	0 0 0 1	0 0 1 0	Rd	Ra	Rm	1	M	0	0	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.171 SMLAWB、SMLAWT

SMLAWB、SMLAWT（符号付き積和演算（ワードとハーフワード）は、符号付き積和演算を実行します。符号付き 32 ビット数値と符号付き 16 ビット数値に対して乗算が実行されます。符号付き 16 ビット数値は、ソースレジスタの下位半分または上位半分から取得されます。2 番目のソースレジスタの他の半分は無視されます。48 ビットの積の上位 32 ビットが 32 ビットの累積値に加算され、結果がデスティネーションレジスタに書き込まれます。48 ビットの積の下位 16 ビットは無視されます。

累積値の加算中にオーバフローが発生すると、APSR の Q フラグがセットされます。乗算中にオーバフローが発生することはありません。

エンコード T1 ARMv6T2、ARMv7

```
SMLAW<y><c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 1 1	Rn	Ra	Rd	0 0 0	M	Rm

```
if Ra == '1111' then SEE SMULWB, SMULWT;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
SMLAW<y><c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
cond	0 0 0 1	0 0 1 0	Rd	Ra	Rm	1	M	0	0	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMLAW<y><c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

where:

<y> Specifies which half of the source register <Rm> is used as the second multiply operand. If <y> is B, then the bottom half (bits [15:0]) of <Rm> is used. If <y> is T, then the top half (bits [31:16]) of <Rm> is used.

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<Rm> The source register whose bottom or top half (selected by <y>) is the second multiply operand.

<Ra> The register that contains the accumulate value.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(R[n]) * SInt(operand2) + (SInt(R[a]) << 16);
    R[d] = result<47:16>;
    if (result >> 16) != SInt(R[d]) then  // Signed overflow
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
SMLAW<y><c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

各項目の説明については以下を参照して下さい。

<y> ソースレジスタ <Rm> のどの半分を 2 番目の乗算オペランドとして使用するかを指定します。<y> が B の場合、<Rm> の下位半分（ビット [15:0]）が使用されます。<y> が T の場合、<Rm> の上位半分（ビット [31:16]）が使用されます。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<Rm> 下位半分または上位半分（<y> によって選択される）が 2 番目の乗算オペランドである、ソースレジスタ

<Ra> 累積値を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(R[n]) * SInt(operand2) + (SInt(R[a]) << 16);
    R[d] = result<47:16>;
    if (result >> 16) != SInt(R[d]) then  // Signed overflow
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.172 SMLSD

Signed Multiply Subtract Dual performs two signed 16 × 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top × bottom and bottom × top multiplication.

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

Encoding T1 ARMv6T2, ARMv7

```
SMLSD{X}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	1 0 0	Rn	Ra	Rd	0 0 0	M	Rm

```
if Ra == '1111' then SEE SMUSD;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_swap = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SMLSD{X}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1	0 0 0 0	Rd	Ra	Rm	0 1	M	1	Rn

```
if Ra == '1111' then SEE SMUSD;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.172 SMLSD

SMLSD（符号付きデュアル積減算）は、2 つの符号付き 16 × 16 ビット乗算を実行します。2 つの積の差を 32 ビット累積オペランドに加算します。

オプションとして、この算術演算を実行する前に 2 番目のオペランドのハーフワード同士を交換することができます。これにより、上位 × 下位、および下位 × 上位の乗算が生成されます。

累積演算中にオーバフローが発生すると、Q フラグがセットされます。乗算中または減算中にオーバフローが発生することはありません。

エンコード T1 ARMv6T2、ARMv7

```
SMLSD{X}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	1 0 0	Rn	Ra	Rd	0 0 0	M	Rm

```
if Ra == '1111' then SEE SMUSD;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_swap = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SMLSD{X}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1	0 0 0 0	Rd	Ra	Rm	0 1	M	1	Rn

```
if Ra == '1111' then SEE SMUSD;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMLSD{X}<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

where:

X	If X is present, the multiplications are bottom × top and top × bottom. If the X is omitted, the multiplications are bottom × bottom and top × top.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The second operand register.
<Ra>	The register that contains the accumulate value.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
SMLSD{X}<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

各項目の説明については以下を参照して下さい。

X	X を指定した場合、乗算は下位 × 上位、および上位 × 下位となります。 X を省略した場合、乗算は下位 × 下位、および上位 × 上位となります。
<c><q>	*標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	2 番目のオペランドレジスタ
<Ra>	累積値を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.173 SMLSLD

Signed Multiply Subtract Long Dual performs two signed 16 × 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top × bottom and bottom × top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

Encoding T1 ARMv6T2, ARMv7

```
SMLSLD{X}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1	0 1 1 1	1	0 1	Rn	RdLo	RdHi	1 1 0	M	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SMLSLD{X}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1 0 1 0 0	RdHi	RdLo	Rm	0 1	M	1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.173 SMLSLD

SMLSLD（符号付きデュアルロング積減算）は、2 つの符号付き 16 × 16 ビット乗算を実行します。2 つの積の差を 64 ビット累積オペランドに加算します。

オプションとして、この算術演算を実行する前に 2 番目のオペランドのハーフワード同士を交換することができます。これにより、上位 × 下位、および下位 × 上位の乗算が生成されます。

この命令の実行中にオーバフローが発生することがありますが、64 ビット加算の結果としてのみ発生します。このオーバフローは、発生しても検出されません。その代わりに、結果がモジュロ 2^{64} にラップアラウンドされます。

エンコード T1 ARMv6T2、ARMv7

```
SMLSLD{X}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1	0 1 1 1	1	0 1	Rn	RdLo	RdHi	1 1 0	M	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SMLSLD{X}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1 0 1 0 0	RdHi	RdLo	Rm	0 1	M	1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMLSLD{X}<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

where:

X — If X is present, the multiplications are bottom × top and top × bottom.
If the X is omitted, the multiplications are bottom × bottom and top × top.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<RdLo> — Supplies the lower 32 bits of the accumulate value, and is the destination register for the lower 32 bits of the result.

<RdHi> — Supplies the upper 32 bits of the accumulate value, and is the destination register for the upper 32 bits of the result.

<Rn> — The first operand register.

<Rm> — The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMLSLD{X}<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

X — X を指定した場合、乗算は下位 × 上位、および上位 × 下位となります。
X を省略した場合、乗算は下位 × 下位、および上位 × 上位となります。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<RdLo> — 加算される値の下位 32 ビットで、同時に結果の下位 32 ビットのデスティネーションレジスタとなります。

<RdHi> — 加算される値の上位 32 ビットで、同時に結果の上位 32 ビットのデスティネーションレジスタとなります。

<Rn> — 最初のオペランドレジスタ

<Rm> — 2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.174 SMMLA

Signed Most Significant Word Multiply Accumulate multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and adds an accumulate value.

Optionally, you can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

Encoding T1 ARMv6T2, ARMv7

```
SMMLA{R}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	1 0 1	Rn	Ra	Rd	0 0 0	R	Rm

```
if Ra == '1111' then SEE SMMUL;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SMMLA{R}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1	0 1 0 1	Rd	Ra	Rm	0 0	R	1	Rn

```
if Ra == '1111' then SEE SMMUL;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.174 SMMLA

SMMLA（符号付き最上位ワード積和演算）は、2 つの符号付き 32 ビット値を乗算し、結果の最上位 32 ビットを抽出し、それを累積値に加算します。

オプションとして、結果を切り捨てるのではなく丸めるように指定することができます。指定した場合、上位ワードが抽出される前に、定数 `0x80000000` が積に加算されます。

エンコード T1 ARMv6T2、ARMv7

```
SMMLA{R}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	1 0 1	Rn	Ra	Rd	0 0 0	R	Rm

```
if Ra == '1111' then SEE SMMUL;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SMMLA{R}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1	0 1 0 1	Rd	Ra	Rm	0 0	R	1	Rn

```
if Ra == '1111' then SEE SMMUL;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMMLA{R}<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

where:

R	If R is present, the multiplication is rounded. If the R is omitted, the multiplication is truncated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The register that contains the first multiply operand.
<Rm>	The register that contains the second multiply operand.
<Ra>	The register that contains the accumulate value.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) + SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMMLA{R}<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

各項目の説明については以下を参照して下さい。

R	R を指定した場合、乗算が丸められます。 R を省略した場合、乗算が切り捨てられます。
<c><q>	*標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初の乗算オペランドを含むレジスタ
<Rm>	2 番目の乗算オペランドを含むレジスタ
<Ra>	累積値を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) + SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.175 SMMLS

Signed Most Significant Word Multiply Subtract multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and subtracts it from an accumulate value.

Optionally, you can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

Encoding T1 ARMv6T2, ARMv7

```
SMMLS{R}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	1 1 0	Rn	Ra	Rd	0 0 0	R	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if BadReg(d) || BadReg(n) || BadReg(m) || BadReg(a) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SMMLS{R}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1	0 1 0 1	Rd	Ra	Rm	1 1	R	1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.175 SMMLS

SMMLS（符号付き最上位ワード積減算）は、2 つの符号付き 32 ビット値を乗算し、結果の最上位 32 ビットを抽出し、それを累積値から減算します。

オプションとして、結果を切り捨てるのではなく丸めるように指定することができます。指定した場合、上位ワードが抽出される前に、定数 `0x80000000` が積に加算されます。

エンコード T1 ARMv6T2、ARMv7

```
SMMLS{R}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	1 1 0	Rn	Ra	Rd	0 0 0	R	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if BadReg(d) || BadReg(n) || BadReg(m) || BadReg(a) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SMMLS{R}<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1	0 1 0 1	Rd	Ra	Rm	1 1	R	1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMMLS{R}<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

where:

R	If R is present, the multiplication is rounded. If the R is omitted, the multiplication is truncated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The register that contains the first multiply operand.
<Rm>	The register that contains the second multiply operand.
<Ra>	The register that contains the accumulate value.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) - SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMMLS{R}<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

各項目の説明については以下を参照して下さい。

R	R を指定した場合、乗算が丸められます。 R を省略した場合、乗算が切り捨てられます。
<c><q>	*標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初の乗算オペランドを含むレジスタ
<Rm>	2 番目の乗算オペランドを含むレジスタ
<Ra>	累積値を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) - SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.176 SMMUL

Signed Most Significant Word Multiply multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and writes those bits to the destination register.

Optionally, you can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

Encoding T1 ARMv6T2, ARMv7

```
SMMUL{R}<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	1 0 1	Rn	1 1 1 1	Rd	0 0 0	R	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  round = (R == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SMMUL{R}<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1	0 1 0 1	Rd	1 1 1 1	Rm	0 0	R	1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.176 SMMUL

SMMUL（符号付き最上位ワード乗算）は、2 つの符号付き 32 ビット値を乗算し、結果の最上位 32 ビットを抽出し、それをデスティネーションレジスタに書き込みます。

オプションとして、結果を切り捨てるのではなく丸めるように指定することができます。指定した場合、上位ワードが抽出される前に、定数 `0x80000000` が積に加算されます。

エンコード T1 ARMv6T2、ARMv7

```
SMMUL{R}<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	1 0 1	Rn	1 1 1 1	Rd	0 0 0	R	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  round = (R == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SMMUL{R}<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1	0 1 0 1	Rd	1 1 1 1	Rm	0 0	R	1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMMUL{R}<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

R	If R is present, the multiplication is rounded. If the R is omitted, the multiplication is truncated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMMUL{R}<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

R	R を指定した場合、乗算が丸められます。 R を省略した場合、乗算が切り捨てられます。
<c><q>	*標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.177 SMUAD

Signed Dual Multiply Add performs two signed 16 × 16-bit multiplications. It adds the products together, and writes the result to the destination register.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top × bottom and bottom × top multiplication.

This instruction sets the Q flag if the addition overflows. The multiplications cannot overflow.

Encoding T1 ARMv6T2, ARMv7

```
SMUAD{X}<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 1 0	Rn	1 1 1 1	Rd	0 0 0	M	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SMUAD{X}<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1 0 0 0 0	Rd	1 1 1 1	Rm	0 0	M	1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.177 SMUAD

SMUAD（符号付きデュアル積加算）は、2 つの符号付き 16 × 16 ビット乗算を実行します。2 つの積を加算し、結果をデスティネーションレジスタに書き込みます。

オプションとして、この算術演算を実行する前に 2 番目のオペランドのハーフワード同士を交換することができます。これにより、上位 × 下位、および下位 × 上位の乗算が生成されます。

加算中にオーバフローが発生すると、Q フラグがセットされます。乗算中にオーバフローが発生することはありません。

エンコード T1 ARMv6T2、ARMv7

```
SMUAD{X}<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 1 0	Rn	1 1 1 1	Rd	0 0 0	M	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SMUAD{X}<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1 0 0 0 0	Rd	1 1 1 1	Rm	0 0	M	1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMUAD{x}<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

X	If X is present, the multiplications are bottom × top and top × bottom. If the X is omitted, the multiplications are bottom × bottom and top × top.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2;
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMUAD{x}<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

X	X を指定した場合、乗算は下位 × 上位、および上位 × 下位となります。 X を省略した場合、乗算は下位 × 下位、および上位 × 上位となります。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2;
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.178 SMULBB, SMULBT, SMULTB, SMULTT

Signed Multiply (halfwords) multiplies two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is written to the destination register. No overflow is possible during this instruction.

Encoding T1 ARMv6T2, ARMv7

```
SMUL<x><y><c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 0 1	Rn	1 1 1 1	Rd	0 0	N	M	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
SMUL<x><y><c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
cond	0 0 0 1 0 1 1 0	Rd	SBZ	Rm	1	M	N	0	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.178 SMULBB、SMULBT、SMULTB、SMULTT

SMULBB、SMULBT、SMULTB、SMULTT（符号付き乗算（ハーフワード））は、それぞれのソースレジスタの下位半分または上位半分から取得された、2 つの符号付き 16 ビット数値を乗算します。これらのソースレジスタの他の半分は無視されます。32 ビットの積がデスティネーションレジスタに書き込まれます。この命令の実行中にオーバフローが発生することはありません。

エンコード T1 ARMv6T2、ARMv7

```
SMUL<x><y><c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 0 1	Rn	1 1 1 1	Rd	0 0	N	M	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
SMUL<x><y><c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
cond	0 0 0 1 0 1 1 0	Rd	SBZ	Rm	1	M	N	0	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMUL<x><y><c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

<x> Specifies which half of the source register <Rn> is used as the first multiply operand. If <x> is B, then the bottom half (bits [15:0]) of <Rn> is used. If <x> is T, then the top half (bits [31:16]) of <Rn> is used.

<y> Specifies which half of the source register <Rm> is used as the second multiply operand. If <y> is B, then the bottom half (bits [15:0]) of <Rm> is used. If <y> is T, then the top half (bits [31:16]) of <Rm> is used.

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The source register whose bottom or top half (selected by <x>) is the first multiply operand.

<Rm> The source register whose bottom or top half (selected by <y>) is the second multiply operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2);
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMUL<x><y><c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<x> ソースレジスタ <Rn> のどの半分を最初の乗算オペランドとして使用するかを指定します。<x> が B の場合、<Rn> の下位半分（ビット [15:0]）が使用されます。<x> が T の場合、<Rn> の上位半分（ビット [31:16]）が使用されます。

<y> ソースレジスタ <Rm> のどの半分を 2 番目の乗算オペランドとして使用するかを指定します。<y> が B の場合、<Rm> の下位半分（ビット [15:0]）が使用されます。<y> が T の場合、<Rm> の上位半分（ビット [31:16]）が使用されます。

<c><q> *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 下位半分または上位半分（<x> によって選択される）が最初の乗算オペランドである、ソースレジスタ

<Rm> 下位半分または上位半分（<y> によって選択される）が 2 番目の乗算オペランドである、ソースレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2);
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.179 SMULL

Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result.

In ARM code, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many processor implementations.

Encoding T1 ARMv6T2, ARMv7

```
SMULL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11 10 9 8 7 6 5 4	3 2 1 0	15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1 0 1 1 1 0 0 0	Rn	RdLo	RdHi	0 0 0 0	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
SMULL{S}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 1 1 0	S	RdHi	RdLo	Rm	1 0 0 1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
if ArchVersion() < 6 && (dHi == n || dLo == n) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.179 SMULL

SMULL（符号付きロング乗算）は、2 つの符号付き 32 ビット値を乗算して、64 ビットの結果を生成します。

ARM コードでは、オプションとして、結果に基づいて条件フラグを更新することができます。このオプションを使用すると、多くのプロセッサ実装で性能が低下します。

エンコード T1 ARMv6T2、ARMv7

```
SMULL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11 10 9 8 7 6 5 4	3 2 1 0	15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1 0 1 1 1 0 0 0	Rn	RdLo	RdHi	0 0 0 0	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
SMULL{S}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 1 1 0	S	RdHi	RdLo	Rm	1 0 0 1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
if ArchVersion() < 6 && (dHi == n || dLo == n) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMULL{S}<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated. S can be specified only for the ARM instruction set.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<RdLo>	Stores the lower 32 bits of the result.
<RdHi>	Stores the upper 32 bits of the result.
<Rn>	The first operand register.
<Rm>	The second operand register.

The pre-UAL syntax SMULL<c>S is equivalent to SMULLS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        APSR.N = result<63>;
        APSR.Z = IsZeroBit(result<63:0>);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
            APSR.V = bit UNKNOWN;
        // else APSR.C, APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMULL{S}<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。 S は、ARM 命令セットでのみ指定できます。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<RdLo>	結果の下位 32 ビットが格納されます。
<RdHi>	結果の上位 32 ビットが格納されます。
<Rn>	最初のオペランドレジスタ
<Rm>	2 番目のオペランドレジスタ

UAL 以前の構文の SMULL<c>S は、SMULLS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        APSR.N = result<63>;
        APSR.Z = IsZeroBit(result<63:0>);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
            APSR.V = bit UNKNOWN;
        // else APSR.C, APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.180 SMULWB, SMULWT

Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

Encoding T1 ARMv6T2, ARMv7

```
SMULW<y><c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 1 1	Rn	1 1 1 1	Rd	0 0 0	M	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
SMULW<y><c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
cond	0 0 0 1 0 0 1 0	Rd	SBZ	Rm	1	M	1	0	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.180 SMULWB、SMULWT

SMULWB、SMULWT（符号付き乗算（ワードとハーフワード））は、符号付き 32 ビット数値と符号付き 16 ビット数値を乗算します。符号付き 16 ビット数値は、ソースレジスタの下位半分または上位半分から取得されます。2 番目のソースレジスタの他の半分は無視されます。48 ビットの積の上位 32 ビットがデスティネーションレジスタに書き込まれます。48 ビットの積の下位 16 ビットは無視されます。この命令の実行中にオーバフローが発生することはありません。

エンコード T1 ARMv6T2、ARMv7

```
SMULW<y><c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1 1	0 1 1 0	0 1 1	Rn	1 1 1 1	Rd	0 0 0	M	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
SMULW<y><c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
cond	0 0 0 1 0 0 1 0	Rd	SBZ	Rm	1	M	1	0	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMULW<y><c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

<y> Specifies which half of the source register <Rm> is used as the second multiply operand. If <y> is B, then the bottom half (bits [15:0]) of <Rm> is used. If <y> is T, then the top half (bits [31:16]) of <Rm> is used.

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<Rm> The source register whose bottom or top half (selected by <y>) is the second multiply operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    product = SInt(R[n]) * SInt(operand2);
    R[d] = product<47:16>;
    // Signed overflow cannot occur
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMULW<y><c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<y> ソースレジスタ <Rm> のどの半分を 2 番目の乗算オペランドとして使用するかを指定します。<y> が B の場合、<Rm> の下位半分（ビット [15:0]）が使用されます。<y> が T の場合、<Rm> の上位半分（ビット [31:16]）が使用されます。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<Rm> 下位半分または上位半分（<y> によって選択される）が 2 番目の乗算オペランドである、ソースレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    product = SInt(R[n]) * SInt(operand2);
    R[d] = product<47:16>;
    // Signed overflow cannot occur
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.181 SMUSD

Signed Dual Multiply Subtract performs two signed 16 × 16-bit multiplications. It subtracts one of the products from the other, and writes the result to the destination register.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top × bottom and bottom × top multiplication.

Overflow cannot occur.

Encoding T1 ARMv6T2, ARMv7

```
SMUSD{X}<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1	0 1 1 0	1 0 0	Rn	1 1 1 1	Rd	0 0 0	M	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SMUSD{X}<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1	0 0 0 0	Rd	1 1 1 1	Rm	0 1	M	1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.181 SMUSD

SMUSD（符号付きデュアル積減算）は、2 つの符号付き 16 × 16 ビット乗算を実行します。一方の積を他方の積から減算し、結果をデスティネーションレジスタに書き込みます。

オプションとして、この算術演算を実行する前に 2 番目のオペランドのハーフワード同士を交換することができます。これにより、上位 × 下位、および下位 × 上位の乗算が生成されます。

オーバフローが発生することはありません。

エンコード T1 ARMv6T2、ARMv7

```
SMUSD{X}<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
1 1 1 1	0 1 1 0	1 0 0	Rn	1 1 1 1	Rd	0 0 0	M	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SMUSD{X}<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 1 1 1	0 0 0 0	Rd	1 1 1 1	Rm	0 1	M	1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMUSD{X}<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

X	If X is present, the multiplications are bottom × top and top × bottom. If the X is omitted, the multiplications are bottom × bottom and top × top.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register.
<Rm>	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2;
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMUSD{X}<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

X	X を指定した場合、乗算は下位 × 上位、および上位 × 下位となります。 X を省略した場合、乗算は下位 × 下位、および上位 × 上位となります。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ
<Rm>	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2;
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.182 SRS

Store Return State is a system instruction. For details see *SRS* on page B6-20.

A8.6.183 SSAT

Signed Saturate saturates an optionally-shifted signed value to a selectable signed range.

The Q flag is set if the operation saturates.

Encoding T1 ARMv6T2, ARMv7

```
SSAT<c> <Rd>,#<imm>,<Rn>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	0	sh	0	Rn				0	imm3			Rd				imm2		(0)		sat_imm			

```
if sh == '1' && (imm3:imm2) == '00000' then SEE SSAT16;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SSAT<c> <Rd>,#<imm>,<Rn>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	sat_imm					Rd				imm5				sh	0	1		Rn			

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.182 SRS

SRS（復帰状態ストア）はシステム命令です。詳細については、*SRS* P. B6-20 を参照して下さい。

A8.6.183 SSAT

SSAT（符号付き飽和）は、オプションとしてシフトされる符号付きの値を、選択可能な符号付き範囲に飽和させます。

演算が飽和した場合は Q フラグがセットされます。

エンコード T1 ARMv6T2、ARMv7

```
SSAT<c> <Rd>,#<imm>,<Rn>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	0	sh	0	Rn				0	imm3			Rd				imm2		(0)		sat_imm			

```
if sh == '1' && (imm3:imm2) == '00000' then SEE SSAT16;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SSAT<c> <Rd>,#<imm>,<Rn>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	sat_imm					Rd				imm5				sh	0	1		Rn			

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SSAT<c><q>  <Rd>, #<imm>, <Rn> {,<shift>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<imm>` The bit position for saturation, in the range 1 to 32.

`<Rn>` The register that contains the value to be saturated.

`<shift>` The optional shift, encoded in the sh bit and five bits in imm3:imm2 for encoding T1 and imm5 for encoding A1. It must be one of:

omitted No shift. Encoded as sh = 0, five bits = `0b00000`

`LSL #<n>` Left shift by `<n>` bits, with `<n>` in the range 1-31.
Encoded as sh = 0, five bits = `<n>`.

`ASR #<n>` Arithmetic right shift by `<n>` bits, with `<n>` in the range 1-31.
Encoded as sh = 1, five bits = `<n>`.

`ASR #32` Arithmetic right shift by 32 bits, permitted only for encoding A1.
Encoded as sh = 1, imm5 = `0b00000`.

Note

An assembler can permit `ASR #0` or `LSL #0` to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, APSR.C);  // APSR.C ignored
    (result, sat) = SignedSatQ(SInt(operand), saturate_to);
    R[d] = SignExtend(result, 32);
    if sat then
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SSAT<c><q>  <Rd>, #<imm>, <Rn> {,<shift>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<imm>` 飽和のビット位置 (1 ～ 32)

`<Rn>` 飽和される値を含むレジスタ

`<shift>` オプションのシフト。sh ビットと、エンコード T1 では imm3:imm2、エンコード A1 では imm5 の 5 ビットにエンコードされます。次のいずれかを使用できます。

省略 シフトなし。sh = 0、5 つのビット = `0b00000` としてエンコードされます。

`LSL #<n>` `<n>` ビットだけ左シフトが適用されます。`<n>` の範囲は 1 ～ 31 です。
sh = 0、5 つのビット = `<n>` としてエンコードされます。

`ASR #<n>` `<n>` ビットだけ算術右シフトが適用されます。`<n>` の範囲は 1 ～ 31 です。
sh = 1、5 つのビット = `<n>` としてエンコードされます。

`ASR #32` 32 ビットだけ算術右シフトが適用されます。エンコード A1 でのみ許可されます。
sh = 1、imm5 = `0b00000` としてエンコードされます。

注

アセンブラは、`ASR #0` または `LSL #0` がシフトの省略と同じ意味にすることを許可できます。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, APSR.C);  // APSR.C ignored
    (result, sat) = SignedSatQ(SInt(operand), saturate_to);
    R[d] = SignExtend(result, 32);
    if sat then
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.184 SSAT16

Signed Saturate 16 saturates two signed 16-bit values to a selected signed range.

The Q flag is set if the operation saturates.

Encoding T1 ARMv6T2, ARMv7

```
SSAT16<c> <Rd>,#<imm>,<Rn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	0	1	0	Rn				0	0	0	0	Rd				0	0	(0)	(0)	sat_imm			

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SSAT16<c> <Rd>,#<imm>,<Rn>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	0	sat_imm				Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rn			

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if d == 15 || n == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.184 SSAT16

SSAT16（符号付き飽和 16）は、2 つの符号付き 16 ビット値を、選択された符号付き範囲に飽和させます。

演算が飽和した場合は Q フラグがセットされます。

エンコード T1 ARMv6T2、ARMv7

```
SSAT16<c> <Rd>,#<imm>,<Rn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	0	1	0	Rn				0	0	0	0	Rd				0	0	(0)	(0)	sat_imm			

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SSAT16<c> <Rd>,#<imm>,<Rn>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	0	sat_imm				Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rn			

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if d == 15 || n == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SSAT16<c><q>  <Rd>, #<imm>, <Rn>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<imm>`	The bit position for saturation, in the range 1 to 16.
`<Rn>`	The register that contains the values to be saturated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = SignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = SignedSatQ(SInt(R[n]<31:16>), saturate_to);
    R[d]<15:0> = SignExtend(result1, 16);
    R[d]<31:16> = SignExtend(result2, 16);
    if sat1 || sat2 then
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SSAT16<c><q>  <Rd>, #<imm>, <Rn>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<imm>`	飽和のビット位置 (1 ～ 16)
`<Rn>`	飽和される値を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = SignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = SignedSatQ(SInt(R[n]<31:16>), saturate_to);
    R[d]<15:0> = SignExtend(result1, 16);
    R[d]<31:16> = SignExtend(result2, 16);
    if sat1 || sat2 then
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.185 SSAX

Signed Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

Encoding T1 ARMv6T2, ARMv7

```
SSAX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 1 0	Rn	1 1 1 1	Rd	0	0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SSAX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 0 0 1	Rn	Rd	(1) (1) (1) (1)	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.185 SSAX

SSAX（交換付き符号付き減算および加算）は、2 番目のオペランドの 2 つのハーフワードを交換し、1 つの 16 ビット整数減算と 1 つの 16 ビット加算を実行して、結果をデスティネーションレジスタに書き込みます。結果に従って APSR.GE ビットをセットします。

エンコード T1 ARMv6T2、ARMv7

```
SSAX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 1 0	Rn	1 1 1 1	Rd	0	0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SSAX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 0 0 1	Rn	Rd	(1) (1) (1) (1)	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SSAX<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<Rm> The second operand register.

The pre-UAL syntax SSUBADDX<c> is equivalent to SSAX<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0>  = sum<15:0>;
    R[d]<31:16> = diff<15:0>;
    APSR.GE<1:0> = if sum  >= 0 then '11' else '00';
    APSR.GE<3:2> = if diff >= 0 then '11' else '00';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SSAX<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<Rm> 2 番目のオペランドレジスタ

UAL 以前の構文の SSUBADDX<c> は、SSAX<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0>  = sum<15:0>;
    R[d]<31:16> = diff<15:0>;
    APSR.GE<1:0> = if sum  >= 0 then '11' else '00';
    APSR.GE<3:2> = if diff >= 0 then '11' else '00';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.186 SSUB16

Signed Subtract 16 performs two 16-bit signed integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

Encoding T1 ARMv6T2, ARMv7

```
SSUB16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SSUB16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 0 0 1	Rn	Rd	(1) (1) (1) (1)	0	1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.186 SSUB16

SSUB16（符号付き減算 16）は、2 つの 16 ビット符号付き整数減算を実行して、結果をデスティネーションレジスタに書き込みます。減算の結果に従って APSR.GE ビットをセットします。

エンコード T1 ARMv6T2、ARMv7

```
SSUB16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SSUB16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 0 0 1	Rn	Rd	(1) (1) (1) (1)	0	1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SSUB16<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<15:0>;
    R[d]<31:16> = diff2<15:0>;
    APSR.GE<1:0> = if diff1 >= 0 then '11' else '00';
    APSR.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SSUB16<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<15:0>;
    R[d]<31:16> = diff2<15:0>;
    APSR.GE<1:0> = if diff1 >= 0 then '11' else '00';
    APSR.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.187 SSUB8

Signed Subtract 8 performs four 8-bit signed integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

Encoding T1 ARMv6T2, ARMv7

```
SSUB8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 0	Rn	1 1 1 1	Rd	0	0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SSUB8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 0 0 0 1	Rn	Rd	(1) (1) (1) (1)	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.187 SSUB8

SSUB8（符号付き減算 8）は、4 つの 8 ビット符号付き整数減算を実行して、結果をデスティネーションレジスタに書き込みます。減算の結果に従って APSR.GE ビットをセットします。

エンコード T1 ARMv6T2、ARMv7

```
SSUB8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 0	Rn	1 1 1 1	Rd	0	0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SSUB8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 0 0 0 1	Rn	Rd	(1) (1) (1) (1)	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SSUB8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0>   = diff1<7:0>;
    R[d]<15:8>  = diff2<7:0>;
    R[d]<23:16> = diff3<7:0>;
    R[d]<31:24> = diff4<7:0>;
    APSR.GE<0>  = if diff1 >= 0 then '1' else '0';
    APSR.GE<1>  = if diff2 >= 0 then '1' else '0';
    APSR.GE<2>  = if diff3 >= 0 then '1' else '0';
    APSR.GE<3>  = if diff4 >= 0 then '1' else '0';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SSUB8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0>   = diff1<7:0>;
    R[d]<15:8>  = diff2<7:0>;
    R[d]<23:16> = diff3<7:0>;
    R[d]<31:24> = diff4<7:0>;
    APSR.GE<0>  = if diff1 >= 0 then '1' else '0';
    APSR.GE<1>  = if diff2 >= 0 then '1' else '0';
    APSR.GE<2>  = if diff3 >= 0 then '1' else '0';
    APSR.GE<3>  = if diff4 >= 0 then '1' else '0';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.188 STC, STC2

Store Coprocessor stores data from a coprocessor to a sequence of consecutive memory addresses. If no coprocessor can execute the instruction, an Undefined Instruction exception is generated.

This is a generic coprocessor instruction. Some of the fields have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer. These fields are the D bit, the CRd field, and in the Unindexed addressing mode only, the imm8 field.

For more information about the coprocessors see *Coprocessor support* on page A2-68.

Encoding T1 / A1 ARMv6T2, ARMv7 for encoding T1
ARMv4*, ARMv5T*, ARMv6*, ARMv7 for encoding A1

```
STC{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm>]{!}
STC{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm>
STC{L}<c> <coproc>,<CRd>,[<Rn>],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	0	Rn				CRd				coproc				imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	P	U	D	W	0	Rn				CRd				coproc				imm8							

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MCRR, MCRR2;
if coproc == '101x' then SEE "Advanced SIMD and VFP";
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

Encoding T2 / A2 ARMv6T2, ARMv7 for encoding T2
ARMv5T*, ARMv6*, ARMv7 for encodingA2

```
STC2{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm>]{!}
STC2{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm>
STC2{L}<c> <coproc>,<CRd>,[<Rn>],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	0	Rn				CRd				coproc				imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	0	Rn				CRd				coproc				imm8							

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MCRR, MCRR2;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

Advanced SIMD and VFP See *Extension register load/store instructions* on page A7-26

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.188 STC、STC2

STC、STC2（コプロセッサストア）は、コプロセッサから連続するメモリアドレスへデータをストアします。どのコプロセッサも命令を実行できない場合は、未定義命令例外が発生します。

これは、汎用のコプロセッサ命令です。一部のフィールドの機能はアーキテクチャで定義されていないため、コプロセッサ命令セットの設計者が自由に使用することができます。これらのフィールドは、D ビット、CRd フィールド、imm8 フィールド（インデクスなしアドレシングモードのみ）です。

コプロセッサの詳細については、*コプロセッサのサポート* P. A2-68 を参照して下さい。

エンコード T1/A1 ARMv6T2、ARMv7（エンコード T1 の場合）
ARMv4*、ARMv5T*、ARMv6*、ARMv7（エンコード A1 の場合）

```
STC{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm>]{!}
STC{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm>
STC{L}<c> <coproc>,<CRd>,[<Rn>],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	0	Rn				CRd				coproc				imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	P	U	D	W	0	Rn				CRd				coproc				imm8							

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MCRR, MCRR2;
if coproc == '101x' then SEE "Advanced SIMD and VFP";
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

エンコード T2/A2 ARMv6T2、ARMv7（エンコード T2 の場合）
ARMv5T*、ARMv6*、ARMv7（エンコード A2 の場合）

```
STC2{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm>]{!}
STC2{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm>
STC2{L}<c> <coproc>,<CRd>,[<Rn>],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	0	Rn				CRd				coproc				imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	0	Rn				CRd				coproc				imm8							

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MCRR, MCRR2;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

アドバンスト SIMD および VFP *拡張機能レジスタロード／ストア命令* P. A7-26 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>{,#+/-<imm>}]`	Offset. P = 1, W = 0.
`STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>,#+/-<imm>]!`	Pre-indexed. P = 1, W = 1.
`STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],#+/-<imm>`	Post-indexed. P = 0, W = 1.
`STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],<option>`	Unindexed. P = 0, W = 0, U = 1.

where:

`2` If specified, selects encoding T2 / A2. If omitted, selects encoding T1 / A1.

`L` If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `STC2` instruction must be unconditional.

`<coproc>` The name of the coprocessor. The standard generic coprocessor names are p0, p1, …, p15.

`<CRd>` The coprocessor source register.

`<Rn>` The base register. The SP can be used. In the ARM instruction set, for offset and unindexed addressing only, the PC can be used. However, use of the PC is deprecated.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. Values are multiples of 4 in the range 0-1020. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of +0.

`<option>` A coprocessor option. An integer in the range 0-255 enclosed in { }. Encoded in imm8.

The pre-UAL syntax `STC<c>L` is equivalent to `STCL<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        NullCheckIfThumbEE(n);
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        repeat
            MemA[address,4] = Coproc_GetWordToStore(cp, ThisInstr());  address = address + 4;
        until Coproc_DoneStoring(cp, ThisInstr());
        if wback then R[n] = offset_addr;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>{,#+/-<imm>}]`	オフセット：P = 1、W = 0
`STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>,#+/-<imm>]!`	プリインデクス：P = 1、W = 1
`STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],#+/-<imm>`	ポストインデクス：P = 0、W = 1
`STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],<option>`	インデクスなし：P = 0、W = 0、U = 1

各項目の説明については以下を参照して下さい。

`2` 指定した場合、エンコード T2/A2 が選択されます。省略した場合、エンコード T1/A1 が選択されます。

`L` 指定した場合、エンコードの D == 1 の形式が選択されます。省略した場合、D == 0 の形式が選択されます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。ARM `STC2` は常に無条件命令です。

`<coproc>` コプロセッサの名前。標準の汎用コプロセッサ名は、p0、p1、...、p15 です。

`<CRd>` コプロセッサのソースレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。ARM 命令では、オフセットアドレシングとインデクスなしのアドレシングに対してのみ PC を指定できます。しかし、PC の使用は推奨されません。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。値は、0 ～ 1020 までの 4 の倍数です。オフセットアドレシング構文では、`<imm>` を省略して、+ 0 のオフセットを指定することができます。

`<option>` コプロセッサオプション。{ } で囲んだ 0 ～ 255 の整数。imm8 にエンコードされます。

UAL 以前の構文の `STC<c>L` は、`STCL<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        NullCheckIfThumbEE(n);
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        repeat
            MemA[address,4] = Coproc_GetWordToStore(cp, ThisInstr());  address = address + 4;
        until Coproc_DoneStoring(cp, ThisInstr());
        if wback then R[n] = offset_addr;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.189 STM / STMIA / STMEA

Store Multiple Increment After (Store Multiple Empty Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

For details of related system instructions see *STM (user registers)* on page B6-22.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7 (not in ThumbEE)

```
STM<c> <Rn>!,<registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	Rn			register_list							

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = TRUE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Encoding T2 ARMv6T2, ARMv7

```
STM<c>.W <Rn>{!},<registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	W	0	Rn				(0)	M	(0)	register_list												

```
n = UInt(Rn);  registers = '0':M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STM<c> <Rn>{!},<registers>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	0	0	1	0	W	0	Rn				register_list															

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

Assembler syntax

```
STM<c><q>  <Rn>{!}, <registers>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rn>` The base register. The SP can be used.

`!` Causes the instruction to write a modified value back to `<Rn>`. Encoded as W = 1.

If ! is omitted, the instruction does not change `<Rn>` in this way. Encoded as W = 0.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.189 STM、STMIA、STMEA

STM、STMIA、STMEA（ポストインクリメント複数ストア（空上昇複数ストア））は、ベースレジスタからのアドレスを使用して、複数のレジスタを連続するメモリ位置にストアします。連続するメモリ位置は、このアドレスから開始されます。オプションとして、ストアされる最後のメモリ位置の次のアドレスをベースアドレスに書き戻すことができます。

関連するシステム命令の詳細については、*STM（ユーザレジスタ）* P. B6- 22 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7（ThumbEE 以外）

```
STM<c> <Rn>!,<registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	Rn			register_list							

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = TRUE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

```
STM<c>.W <Rn>{!},<registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	W	0	Rn				(0)	M	(0)	register_list												

```
n = UInt(Rn);  registers = '0':M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STM<c> <Rn>{!},<registers>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	0	0	1	0	W	0	Rn				register_list															

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

アセンブラ構文

```
STM<c><q>  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`!` 命令の実行によって変更された値を `<Rn>` に書き戻すように指示します。W = 1 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<registers>` Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address.

Encoding T2 does not support a list containing only one register. If an STM instruction with just one register <Rt> in the list is assembled to Thumb and encoding T1 is not available, it is assembled to the equivalent STR<c><q> <Rt>,[<Rn>]{,#4} instruction.

The SP and PC can be in the list in ARM code, but not in Thumb code. However, ARM instructions that include the SP or the PC in the list are deprecated.

Encoding T2 is not available for instructions with the base register in the list and ! specified, and the use of such instructions is deprecated. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

STMEA and STMIA are pseudo-instructions for STM. STMEA refers to its use for pushing data onto Empty Ascending stacks.

The pre-UAL syntaxes STM<c>IA and STM<c>EA are equivalent to STM<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;  // Only possible for encodings T1 and A1
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] + 4*BitCount(registers);
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

! を省略した場合は、命令によって <Rn> は変更されません。W = 0 としてエンコードされます。

`<registers>` ストアされる 1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリスト。番号が最も小さいレジスタが最下位のメモリアドレスにストアされ、以下順にストアが行われ、番号が最も大きいレジスタが最上位のメモリアドレスにストアされます。

エンコード T2 では、1 つのレジスタしか含まないリストはサポートされません。リスト内に <Rt> のみを含む STM 命令が Thumb にアセンブルされ、エンコード T1 が使用できない場合は、その命令と等価な STR<c><q> <Rt>,[<Rn>]{,#4} 命令にアセンブルされます。

ARM コードではリストに SP と PC を含めることができますが、Thumb コードではできません。しかし、リストに SP または PC を含める ARM 命令は推奨されません。

ベースレジスタがリストに含まれ、! が指定されている命令では、エンコード T2 を使用できません。このような命令の使用は推奨されません。ベースレジスタが、リスト内の番号が最も小さいレジスタではない場合、このような命令はベースレジスタの未知の値をストアします。

STMEA および STMIA は、STM の擬似命令です。STMEA は、空上昇スタックにデータをプッシュするために使用されることを示します。

UAL 以前の構文の STMSTM<c>IA および STM<c>EA は、STM<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;  // Only possible for encodings T1 and A1
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] + 4*BitCount(registers);
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.190 STMDA / STMED

Store Multiple Decrement After (Store Multiple Empty Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

For details of related system instructions see *STM (user registers)* on page B6-22.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STMDA<c> <Rn>{!},<registers>
```

31 30 29 28	27 26 25 24 23 22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0 0 0 0	W	0	Rn	register_list

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.190 STMDA、STMED

STMDA、STMED（ポストデクリメント複数ストア（空下降複数ストア））は、ベースレジスタからのアドレスを使用して、複数のレジスタを連続するメモリ位置にストアします。連続するメモリ位置は、このアドレスで終了します。オプションとして、ロードされる最下位メモリ位置の 1 つ前のアドレスをベースレジスタに書き戻すことができます。

関連するシステム命令の詳細については、*STM（ユーザレジスタ）* P. B6- 22 を参照して下さい。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STMDA<c> <Rn>{!},<registers>
```

31 30 29 28	27 26 25 24 23 22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0 0 0 0	W	0	Rn	register_list

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
STMDA<c><q>  <Rn>{!}, <registers>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The base register. The SP can be used.

! Causes the instruction to write a modified value back to <Rn>. Encoded as W = 1.

If ! is omitted, the instruction does not change <Rn> in this way. Encoded as W = 0.

<registers> Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address.

The SP and PC can be in the list. However, instructions that include the SP or the PC in the list are deprecated.

The use of instructions with the base register in the list and ! specified is deprecated. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

STMED is s pseudo-instruction for STMDA, referring to its use for pushing data onto Empty Descending stacks.

The pre-UAL syntaxes STM<c>DA and STM<c>ED are equivalent to STMDA<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] - 4*BitCount(registers);
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
STMDA<c><q>  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> ベースレジスタ。このレジスタには SP を指定できます。

! 命令の実行によって変更された値を <Rn> に書き戻すように指示します。W = 1 としてエンコードされます。

! を省略した場合は、命令によって <Rn> は変更されません。W = 0 としてエンコードされます。

<registers> ストアされる 1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリスト。番号が最も小さいレジスタが最下位のメモリアドレスにストアされ、以下順にストアが行われ、番号が最も大きいレジスタが最上位のメモリアドレスにストアされます。

リストに SP と PC を含めることができます。しかし、リストに SP または PC を含める命令は推奨されません。

ベースレジスタがリストに含まれ、! が指定されている命令の使用は推奨されません。ベースレジスタが、リスト内の番号が最も小さいレジスタではない場合、このような命令はベースレジスタの未知の値をストアします。

STMED は STMDA の擬似命令で、空下降スタックにデータをプッシュするために使用されることを示します。

UAL 以前の構文の STM<c>DA および STM<c>ED は、STMDA<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] - 4*BitCount(registers);
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.191 STMDB / STMFD

Store Multiple Decrement Before (Store Multiple Full Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

For details of related system instructions see *STM (user registers)* on page B6-22.

Encoding T1 ARMv6T2, ARMv7

```
STMDB<c> <Rn>{!},<registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	0	0	W	0	Rn				(0)	M	(0)	register_list												

```
if W == '1' && Rn == '1101' then SEE PUSH;
n = UInt(Rn);  registers = '0':M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STMDB<c> <Rn>{!},<registers>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	0	1	0	0	W	0	Rn				register_list															

```
if W == '1' && Rn == '1101' && BitCount(register_list) >= 2 then SEE PUSH;
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

Assembler syntax

```
STMDB<c><q>  <Rn>{!}, <registers>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rn>` The base register. The SP can be used. If it is the SP and ! is specified, it is treated as described in *PUSH* on page A8-248.

`!` Causes the instruction to write a modified value back to `<Rn>`. Encoded as W = 1.

If ! is omitted, the instruction does not change `<Rn>` in this way. Encoded as W = 0.

`<registers>` Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address.

Encoding T1 does not support a list containing only one register. If an `STMDB` instruction with just one register `<Rt>` in the list is assembled to Thumb, it is assembled to the equivalent `STR<c><q> <Rt>,[<Rn>,#-4]{!}` instruction.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.191 STMDB、STMFD

STMDB、STMFD（プリデクリメント複数ストア（フル下降複数ストア））は、ベースレジスタからのアドレスを使用して、複数のレジスタを連続するメモリ位置にストアします。連続するメモリ位置は、このアドレスの 1 つ前で終了します。オプションとして、ストアされる最初のメモリ位置のアドレスをベースアドレスに書き戻すことができます。

関連するシステム命令の詳細については、*STM（ユーザレジスタ）* P. B6- 22 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
STMDB<c> <Rn>{!},<registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	0	0	W	0	Rn				(0)	M	(0)	register_list												

```
if W == '1' && Rn == '1101' then SEE PUSH;
n = UInt(Rn);  registers = '0':M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STMDB<c> <Rn>{!},<registers>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	0	1	0	0	W	0	Rn				register_list															

```
if W == '1' && Rn == '1101' && BitCount(register_list) >= 2 then SEE PUSH;
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

アセンブラ構文

```
STMDB<c><q>  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。SP と ! を指定した場合は、*PUSH* P. A8- 248 に説明されているように扱われます。

`!` 命令の実行によって変更された値を `<Rn>` に書き戻すように指示します。W = 1 としてエンコードされます。

! を省略した場合は、命令によって `<Rn>` は変更されません。W = 0 としてエンコードされます。

`<registers>` ストアされる 1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリスト。番号が最も小さいレジスタが最下位のメモリアドレスにストアされ、以下順にストアが行われ、番号が最も大きいレジスタが最上位のメモリアドレスにストアされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The SP and PC can be in the list in ARM code, but not in Thumb code. However, ARM instructions that include the SP or the PC in the list are deprecated.

Instructions with the base register in the list and ! specified are only available in the ARM instruction set, and the use of such instructions is deprecated. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

STMFD is a pseudo-instruction for STMDB, referring to its use for pushing data onto Full Descending stacks.

The pre-UAL syntaxes STM<c>DB and STM<c>FD are equivalent to STMDB<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;  // Only possible for encoding A1
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] - 4*BitCount(registers);
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

エンコード T1 では、1 つのレジスタのみを含むリストはサポートされません。リストに <Rt> のみを含む STMDB 命令が Thumb にアセンブルされた場合、その命令と等価な STR<c><q> <Rt>,[<Rn>,#-4]{!} 命令にアセンブルされます。

ARM コードではリストに SP と PC を含めることができますが、Thumb コードではできません。しかし、リストに SP または PC を含める ARM 命令は推奨されません。

ベースレジスタがリストに含まれ、! が指定されている命令は、ARM 命令セットでのみ使用できます。このような命令の使用は推奨されません。ベースレジスタが、リスト内の番号が最も小さいレジスタではない場合、このような命令はベースレジスタの未知の値をストアします。

STMFD は STMDB の擬似命令で、フル下降スタックにデータをプッシュするために使用されることを示します。

UAL 以前の構文の STM<c>DB および STM<c>FD は、STMDB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;  // Only possible for encoding A1
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] - 4*BitCount(registers);
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.192 STMIB / STMFA

Store Multiple Increment Before (Store Multiple Full Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

For details of related system instructions see *STM (user registers)* on page B6-22.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STMIB<c> <Rn>{!},<registers>
```

31 30 29 28	27 26 25 24 23 22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0 1 1 0	W	0	Rn	register_list

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.192 STMIB、STMFA

STMIB、STMFA（プリインクリメント複数ストア（フル上昇複数ストア））は、ベースレジスタからのアドレスを使用して、複数のレジスタを連続するメモリ位置にストアします。連続するメモリ位置は、このアドレスの 1 つ後から開始されます。オプションとして、ロードされる最後のメモリ位置のアドレスをベースレジスタに書き戻すことができます。

関連するシステム命令の詳細については、*STM（ユーザレジスタ）* P. B6- 22 を参照して下さい。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STMIB<c> <Rn>{!},<registers>
```

31 30 29 28	27 26 25 24 23 22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0 1 1 0	W	0	Rn	register_list

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
STMIB<c><q>  <Rn>{!}, <registers>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The base register. The SP can be used.

! Causes the instruction to write a modified value back to <Rn>. Encoded as W = 1.

If ! is omitted, the instruction does not change <Rn> in this way. Encoded as W = 0.

<registers> Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address.

The SP and PC can be in the list. However, instructions that include the SP or the PC in the list are deprecated.

The use of instructions with the base register in the list and ! specified is deprecated. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

STMFA is a pseudo-instruction for STMIB, referring to its use for pushing data onto Full Ascending stacks.

The pre-UAL syntax STM<c>IB and STM<c>FA are equivalent to STMIB<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] + 4*BitCount(registers);
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
STMIB<c><q>  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> ベースレジスタ。このレジスタには SP を指定できます。

! 命令の実行によって変更された値を <Rn> に書き戻すように指示します。W = 1 としてエンコードされます。

! を省略した場合は、命令によって <Rn> は変更されません。W = 0 としてエンコードされます。

<registers> ストアされる 1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリスト。番号が最も小さいレジスタが最下位のメモリアドレスにストアされ、以下順にストアが行われ、番号が最も大きいレジスタが最上位のメモリアドレスにストアされます。

リストに SP と PC を含めることができます。しかし、リストに SP または PC を含める命令は推奨されません。

ベースレジスタがリストに含まれ、! が指定されている命令の使用は推奨されません。ベースレジスタが、リスト内の番号が最も小さいレジスタではない場合、このような命令はベースレジスタの未知の値をストアします。

STMFA は STMIB の擬似命令で、フル上昇スタックにデータをプッシュするために使用されることを示します。

UAL 以前の構文の STM<c>IB および STM<c>FA は、STMIB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;
            else
                MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then
        MemA[address,4] = PCStoreValue();
    if wback then R[n] = R[n] + 4*BitCount(registers);
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.193 STR (immediate, Thumb)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
STR<c> <Rt>, [<Rn>{,#<imm>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
STR<c> <Rt>,[SP,#<imm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rt			imm8							

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T3 ARMv6T2, ARMv7

```
STR<c>.W <Rt>,[<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	1	0	0	Rn				Rt				imm12											

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv6T2, ARMv7

```
STR<c> <Rt>,[<Rn>,#-<imm8>]
STR<c> <Rt>,[<Rn>],#+/-<imm8>
STR<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	0	Rn				Rt				1	P	U	W	imm8							

```
if P == '1' && U == '1' && W == '0' then SEE STRT;
if Rn == '1101' && P == '1' && U == '0' && W == '1' && imm8 == '00000100' then SEE PUSH;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.193 STR（イミディエート、Thumb）

STR（レジスタストア（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 ワードをストアします。オフセット アドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
STR<c> <Rt>, [<Rn>{,#<imm>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
STR<c> <Rt>,[SP,#<imm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rt			imm8							

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T3 ARMv6T2、ARMv7

```
STR<c>.W <Rt>,[<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	1	0	0	Rn				Rt				imm12											

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE;
```

エンコード T4 ARMv6T2、ARMv7

```
STR<c> <Rt>,[<Rn>,#-<imm8>]
STR<c> <Rt>,[<Rn>],#+/-<imm8>
STR<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	0	Rn				Rt				1	P	U	W	imm8							

```
if P == '1' && U == '1' && W == '0' then SEE STRT;
if Rn == '1101' && P == '1' && U == '0' && W == '1' && imm8 == '00000100' then SEE PUSH;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STR<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`STR<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STR<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register. The SP can be used.

`<Rn>` The base register. The SP can be used.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. Values are:

Encoding T1	multiples of 4 in the range 0-124
Encoding T2	multiples of 4 in the range 0-1020
Encoding T3	any value in the range 0-4095
Encoding T4	any value in the range 0-255.

For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if UnalignedSupport() || address<1:0> == '00' then
        MemU[address,4] = R[t];
    else // Can only occur before ARMv7
        MemU[address,4] = bits(32) UNKNOWN;
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

ThumbEE instruction

ThumbEE has an additional `STR` (immediate) encoding. For details see *STR (immediate)* on page A9-21.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`STR<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`STR<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`STR<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` ソースレジスタ。このレジスタには SP を指定できます。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。値は次のとおりです。

エンコード T1	4 の倍数で 0 ～ 124
エンコード T2	4 の倍数で 0 ～ 1020
エンコード T3	0 ～ 4095 の任意の値
エンコード T4	0 ～ 255 の任意の値

オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if UnalignedSupport() || address<1:0> == '00' then
        MemU[address,4] = R[t];
    else // Can only occur before ARMv7
        MemU[address,4] = bits(32) UNKNOWN;
    if wback then R[n] = offset_addr;
```

例外

データアボート

ThumbEE 命令

ThumbEE には、追加の `STR`（イミディエート）エンコードがあります。詳細については、*STR（イミディエート）* P. A9-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.194 STR (immediate, ARM)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STR<c> <Rt>,[<Rn>{,#+/-<imm12>}]
STR<c> <Rt>,[<Rn>],#+/-<imm12>
STR<c> <Rt>,[<Rn>,#+/-<imm12>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	P	U	0	W	0	Rn	Rt	imm12

```
if P == '0' && W == '1' then SEE STRT;
if Rn == '1101' && P == '1' && U == '0' && W == '1' && imm12 == '000000000100' then SEE PUSH;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.194 STR（イミディエート、ARM）

STR（レジスタストア（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 ワードをストアします。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STR<c> <Rt>,[<Rn>{,#+/-<imm12>}]
STR<c> <Rt>,[<Rn>],#+/-<imm12>
STR<c> <Rt>,[<Rn>,#+/-<imm12>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	P	U	0	W	0	Rn	Rt	imm12

```
if P == '0' && W == '1' then SEE STRT;
if Rn == '1101' && P == '1' && U == '0' && W == '1' && imm12 == '000000000100' then SEE PUSH;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STR<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`STR<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STR<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register. The SP or the PC can be used. However, use of the PC is deprecated.

`<Rn>` The base register. The SP can be used. For offset addressing only, the PC can be used. However, use of the PC is deprecated.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. Any value in the range 0-4095 is permitted. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,4] = if t == 15 then PCStoreValue() else R[t];
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`STR<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`STR<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`STR<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` ソースレジスタ。SP または PC を指定できます。しかし、PC の使用は推奨されません。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC は、オフセットアドレシングでのみ指定できます。しかし、PC の使用は推奨されません。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合 (`add == FALSE`) は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。0 ～ 4095 の任意の値が許可されます。オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,4] = if t == 15 then PCStoreValue() else R[t];
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.195 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
STR<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	0 0 0	Rm	Rn	Rt

```
if CurrentInstrSet() == InstrSet_ThumbEE then SEE "Modified operation in ThumbEE";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
STR<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12	11 10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8 7 6	5 4 3 2 1 0
1 1 1 1 1	0 0	0	0	1 0	0	Rn	Rt	0	0 0 0 0 0	imm2 Rm

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STR<c> <Rt>,[<Rn>,+/-<Rm>{, <shift>}]{!}
STR<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 1 1	P	U	0	W	0	Rn	Rt	imm5	type	0	Rm

```
if P == '0' && W == '1' then SEE STRT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

Modified operation in ThumbEE See *STR (register)* on page A9-12

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.195 STR（レジスタ）

STR（レジスタストア（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、レジスタからそのメモリに 1 ワードをストアします。オフセットレジスタの値は、オプションとしてシフトすることができます。メモリアクセスの詳細については、*メモリアクセス*P. A8- 13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
STR<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	0 0 0	Rm	Rn	Rt

```
if CurrentInstrSet() == InstrSet_ThumbEE then SEE "Modified operation in ThumbEE";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
STR<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12	11 10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8 7 6	5 4 3 2 1 0
1 1 1 1 1	0 0	0	0	1 0	0	Rn	Rt	0	0 0 0 0 0	imm2 Rm

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STR<c> <Rt>,[<Rn>,+/-<Rm>{, <shift>}]{!}
STR<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 1 1	P	U	0	W	0	Rn	Rt	imm5	type	0	Rm

```
if P == '0' && W == '1' then SEE STRT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

ThumbEE で変更された動作 *STR（レジスタ）*P. A9- 12 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STR<c><q> <Rt>, [<Rn>, <Rm>{, <shift>}]`	Offset: `index==TRUE`, `wback==FALSE`
`STR<c><q> <Rt>, [<Rn>, <Rm>{, <shift>}]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STR<c><q> <Rt>, [<Rn>], <Rm>{, <shift>}`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register. The SP can be used. In the ARM instruction set, the PC can be used. However, use of the PC is deprecated.

`<Rn>` The base register. The SP can be used. In the ARM instruction set, for offset addressing only, the PC can be used. However, use of the PC is deprecated.

`+/-` Is + or omitted if the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, add == FALSE).

`<Rm>` Contains the offset that is optionally shifted and added to the value of `<Rn>` to form the address.

`<shift>` The shift to apply to the value read from `<Rm>`. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted. For encoding T2, `<shift>` can only be omitted, encoded as imm2 = `0b00`, or `LSL #<imm>` with `<imm>` = 1, 2, or 3, and `<imm>` encoded in imm2. For encoding A1, see *Shifts applied to a register* on page A8-10.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    if t == 15 then  // Only possible for encoding A1
        data = PCStoreValue();
    else
        data = R[t];
    if UnalignedSupport() || address<1:0> == '00' || CurrentInstrSet() == InstrSet_ARM then
        MemU[address,4] = data;
    else // Can only occur before ARMv7
        MemU[address,4] = bits(32) UNKNOWN;
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`STR<c><q> <Rt>, [<Rn>, <Rm>{, <shift>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`STR<c><q> <Rt>, [<Rn>, <Rm>{, <shift>}]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`STR<c><q> <Rt>, [<Rn>], <Rm>{, <shift>}`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` ソースレジスタ。このレジスタには SP を指定できます。ARM 命令セットでは、PC を指定できます。しかし、PC の使用は推奨されません。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。ARM 命令では、オフセットアドレシングに対してのみ PC を指定できます。しかし、PC の使用は推奨されません。

`+/-` オプションでシフトされる `<Rm>` の値をベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、add == FALSE）は、「-」にします。

`<Rm>` アドレス生成のためにオプションでシフトされて、`<Rn>` の値に加算されるオフセット

`<shift>` `<Rm>` から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。エンコード T2 では、`<shift>` に省略（imm2 = `0b00` としてエンコードされます）または `LSL #<imm>`（`<imm>` = 1、2、3 のいずれか、`<imm>` は imm2 にエンコードされます）のみ使用できます。エンコード A1 については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    if t == 15 then  // Only possible for encoding A1
        data = PCStoreValue();
    else
        data = R[t];
    if UnalignedSupport() || address<1:0> == '00' || CurrentInstrSet() == InstrSet_ARM then
        MemU[address,4] = data;
    else // Can only occur before ARMv7
        MemU[address,4] = bits(32) UNKNOWN;
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.196 STRB (immediate, Thumb)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
STRB<c> <Rt>,[<Rn>,#<imm5>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 ARMv6T2, ARMv7

```
STRB<c>.W <Rt>,[<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	0	0	Rn				Rt				imm12											

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if BadReg(t) then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

```
STRB<c> <Rt>,[<Rn>,#-<imm8>]
STRB<c> <Rt>,[<Rn>],#+/-<imm8>
STRB<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	Rn				Rt				1	P	U	W	imm8							

```
if P == '1' && U == '1' && W == '0' then SEE STRBT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.196 STRB（イミディエート、Thumb）

STRB（レジスタストア バイト（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 バイトをストアします。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス*P. A8-13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
STRB<c> <Rt>,[<Rn>,#<imm5>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T2 ARMv6T2、ARMv7

```
STRB<c>.W <Rt>,[<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	0	0	Rn				Rt				imm12											

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if BadReg(t) then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

```
STRB<c> <Rt>,[<Rn>,#-<imm8>]
STRB<c> <Rt>,[<Rn>],#+/-<imm8>
STRB<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	Rn				Rt				1	P	U	W	imm8							

```
if P == '1' && U == '1' && W == '0' then SEE STRBT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STRB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`STRB<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STRB<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register.

`<Rn>` The base register. The SP can be used.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. Values are:

Encoding T1 any value in the range 0-31

Encoding T2 any value in the range 0-4095

Encoding T3 any value in the range 0-255.

For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `STR<c>B` is equivalent to `STRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`STRB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`STRB<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`STRB<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` ソースレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。値は次のとおりです。

エンコード T1 0 ～ 31 の任意の値

エンコード T2 0 ～ 4095 の任意の値

エンコード T3 0 ～ 255 の任意の値

オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の `STR<c>B` は、`STRB<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.197 STRB (immediate, ARM)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STRB<c> <Rt>,[<Rn>{,#+/-<imm12>}]
STRB<c> <Rt>,[<Rn>],#+/-<imm12>
STRB<c> <Rt>,[<Rn>,#+/-<imm12>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	P	U	1	W	0	Rn	Rt	imm12

```
if P == '0' && W == '1' then SEE STRBT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.197 STRB（イミディエート、ARM）

STRB（レジスタストア バイト（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 バイトをストアします。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス*P. A8- 13 を参照して下さい。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STRB<c> <Rt>,[<Rn>{,#+/-<imm12>}]
STRB<c> <Rt>,[<Rn>],#+/-<imm12>
STRB<c> <Rt>,[<Rn>,#+/-<imm12>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	P	U	1	W	0	Rn	Rt	imm12

```
if P == '0' && W == '1' then SEE STRBT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STRB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`STRB<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STRB<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register.

`<Rn>` The base register. The SP can be used. For offset addressing only, the PC can be used. However, use of the PC is deprecated.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. Values are 0-4095. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `STR<c>B` is equivalent to `STRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`STRB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`STRB<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`STRB<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` ソースレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC は、オフセットアドレシングでのみ指定できます。しかし、PC の使用は推奨されません。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。値は 0 ～ 4095 です。オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の `STR<c>B` は、`STRB<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.198 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
STRB<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	0 1 0	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
STRB<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	15 14 13 12	11	10 9 8 7 6 5 4	3 2 1 0
1 1 1 1 1 0 0 0 0 0 0 0 0	Rn	Rt	0 0 0 0 0	imm2	Rm

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(t) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STRB<c> <Rt>,[<Rn>,+/-<Rm>{, <shift>}]{!}
STRB<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 1 1	P	U	1	W	0	Rn	Rt	imm5	type	0	Rm

```
if P == '0' && W == '1' then SEE STRBT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.198 STRB（レジスタ）

STRB（レジスタストア バイト（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、レジスタからそのメモリに 1 バイトをストアします。オフセットレジスタの値は、オプションとしてシフトすることができます。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
STRB<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	0 1 0	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
STRB<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	15 14 13 12	11	10 9 8 7 6 5 4	3 2 1 0
1 1 1 1 1 0 0 0 0 0 0 0 0	Rn	Rt	0 0 0 0 0	imm2	Rm

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(t) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STRB<c> <Rt>,[<Rn>,+/-<Rm>{, <shift>}]{!}
STRB<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 1 1	P	U	1	W	0	Rn	Rt	imm5	type	0	Rm

```
if P == '0' && W == '1' then SEE STRBT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STRB<c><q> <Rt>, [<Rn>, <Rm>{, <shift>}]`	Offset: `index==TRUE`, `wback==FALSE`
`STRB<c><q> <Rt>, [<Rn>, <Rm>{, <shift>}]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STRB<c><q> <Rt>, [<Rn>], <Rm>{, <shift>}`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register.

`<Rn>` The base register. The SP can be used. In the ARM instruction set, for offset addressing only, the PC can be used. However, use of the PC is deprecated.

`+/-` Is + or omitted if the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, add == FALSE).

`<Rm>` Contains the offset that is optionally shifted and added to the value of `<Rn>` to form the address.

`<shift>` The shift to apply to the value read from `<Rm>`. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted. For encoding T2, `<shift>` can only be omitted, encoded as imm2 = `0b00`, or `LSL #<imm>` with `<imm>` = 1, 2, or 3, and `<imm>` encoded in imm2. For encoding A1, see *Shifts applied to a register* on page A8-10.

The pre-UAL syntax `STR<c>B` is equivalent to `STRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`STRB<c><q> <Rt>, [<Rn>, <Rm>{, <shift>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`STRB<c><q> <Rt>, [<Rn>, <Rm>{, <shift>}]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`STRB<c><q> <Rt>, [<Rn>], <Rm>{, <shift>}`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` ソースレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。ARM 命令セットでは、オフセットアドレシングに対してのみ PC を指定できます。しかし、PC の使用は推奨されません。

`+/-` オプションとしてシフトされる `<Rm>` の値をベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、add == FALSE）は、「-」にします。

`<Rm>` アドレス生成のためにオプションとしてシフトされ、`<Rn>` の値に加算されるオフセット

`<shift>` `<Rm>` から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。エンコード T2 では、`<shift>` に省略（imm2 = `0b00` としてエンコードされます）または `LSL #<imm>`（`<imm>` = 1、2、3 のいずれか、`<imm>` は imm2 にエンコードされます）のみ使用できます。エンコード A1 については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

UAL 以前の構文の `STR<c>B` は、`STRB<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.199 STRBT

Store Register Byte Unprivileged and stores a byte from a register to memory. For information about memory accesses see *Memory accesses* on page A8-13.

The memory access is restricted as if the processor were running in User mode. (This makes no difference if the processor is actually running in User mode.)

The Thumb instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The ARM instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

Encoding T1 ARMv6T2, ARMv7

```
STRBT<c> <Rt>,[<Rn>,#<imm8>]
```

15 14 13 12 11	10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 1	0 0	0	0	0 0	0	Rn	Rt	1	1 1 0	imm8

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STRBT<c> <Rt>,[<Rn>],#+/-<imm12>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	0	U	1	1	0	Rn	Rt	imm12

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

Encoding A2 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STRBT<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 1 1	0	U	1	1	0	Rn	Rt	imm5	type	0	Rm

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && m == n then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.199 STRBT

STRBT（非特権レジスタストア バイト）は、レジスタからメモリに 1 バイトをストアします。メモリアクセスの詳細については、*メモリアクセス*P. A8- 13 を参照して下さい。

プロセッサがユーザモードで実行されている場合と同様にメモリアクセスが制限されます（実際にプロセッサがユーザモードで実行されている場合は、何も変わりません）。

Thumb 命令では、オフセット アドレシングモードを使用します。このモードでは、メモリアクセスに使用されるアドレスを、ベースレジスタの値とイミディエートオフセットから計算し、ベースレジスタの値は変更しません。

ARM 命令では、ポストインデクス アドレシングモードを使用します。このモードでは、ベースレジスタの値をメモリアクセス用のアドレスとして使用し、ベースレジスタの値とオフセットから新しいアドレスを計算して、ベースレジスタに書き戻します。オフセットは、イミディエート値またはオプションとしてシフトされるレジスタの値です。

エンコード T1 ARMv6T2、ARMv7

```
STRBT<c> <Rt>,[<Rn>,#<imm8>]
```

15 14 13 12 11	10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 1	0 0	0	0	0 0	0	Rn	Rt	1	1 1 0	imm8

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STRBT<c> <Rt>,[<Rn>],#+/-<imm12>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 1 0	0	U	1	1	0	Rn	Rt	imm12

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

エンコード A2 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STRBT<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 1 1	0	U	1	1	0	Rn	Rt	imm5	type	0	Rm

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(type, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && m == n then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STRBT<c><q> <Rt>, [<Rn> {, #<imm>}]`	Offset: Thumb only
`STRBT<c><q> <Rt>, [<Rn>] {, #<imm>}`	Post-indexed: ARM only
`STRBT<c><q> <Rt>, [<Rn>], +/-<Rm> {, <shift>}`	Post-indexed: ARM only

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register.

`<Rn>` The base register. The SP can be used.

`+/-` Is + or omitted if `<imm>` or the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, `add == FALSE`).

`<imm>` The immediate offset applied to the value of `<Rn>`. Values are 0-255 for encoding T1, and 0-4095 for encoding A1. `<imm>` can be omitted, meaning an offset of 0.

`<Rm>` Contains the offset that is optionally shifted and added to the value of `<Rn>` to form the address.

`<shift>` The shift to apply to the value read from `<Rm>`. If omitted, no shift is applied. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

The pre-UAL syntax `STR<c>BT` is equivalent to `STRBT<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then Shift(R[m], shift_t, shift_n, APSR.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    MemU_unpriv[address,1] = R[t]<7:0>;
    if postindex then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`STRBT<c><q> <Rt>, [<Rn> {, #<imm>}]`	オフセット：Thumb のみ
`STRBT<c><q> <Rt>, [<Rn>] {, #<imm>}`	ポストインデクス：ARM のみ
`STRBT<c><q> <Rt>, [<Rn>], +/-<Rm> {, <shift>}`	ポストインデクス：ARM のみ

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>` ソースレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`+/-` `<imm>` またはオプションとしてシフトされる `<Rm>` の値をベースレジスタの値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、`add == FALSE`）は、「-」にします。

`<imm>` `<Rn>` の値に適用されるイミディエートオフセット。値の範囲は、エンコード T1 では 0 〜 255、エンコード A1 では 0 〜 4095 です。`<imm>` は省略可能で、その場合オフセットは 0 と見なされます。

`<Rm>` アドレス生成のためにオプションとしてシフトされ、`<Rn>` の値に加算されるオフセット

`<shift>` `<Rm>` から読み出された値に適用されるシフト。省略した場合、シフトは適用されません。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8- 10 を参照して下さい。

UAL 以前の構文の `STR<c>BT` は、`STRBT<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then Shift(R[m], shift_t, shift_n, APSR.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    MemU_unpriv[address,1] = R[t]<7:0>;
    if postindex then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.200 STRD (immediate)

Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
STRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm>}]
STRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm>
STRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm>]!
```

15 14 13 12 11	10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0 1	0 0	P	U	1	W	0	Rn	Rt	Rt2	imm8

```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if n == 15 || BadReg(t) || BadReg(t2) then UNPREDICTABLE;
```

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
STRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm8>}]
STRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm8>
STRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm8>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	1	W	0	Rn	Rt	imm4H	1 1 1 1	imm4L

```
if Rt<0> == '1' then UNDEFINED;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;
```

Related encodings See *Load/store dual, load/store exclusive, table branch* on page A6-24

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.200 STRD（イミディエート）

STRD（デュアル レジスタストア（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、2 つのレジスタからそのメモリに 2 ワードをストアします。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス*P. A8-13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
STRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm>}]
STRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm>
STRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm>]!
```

15 14 13 12 11	10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0 1	0 0	P	U	1	W	0	Rn	Rt	Rt2	imm8

```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if n == 15 || BadReg(t) || BadReg(t2) then UNPREDICTABLE;
```

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
STRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm8>}]
STRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm8>
STRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm8>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	1	W	0	Rn	Rt	imm4H	1 1 1 1	imm4L

```
if Rt<0> == '1' then UNDEFINED;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;
```

関連エンコード *デュアルロード／ストア*、*排他ロード／ストア*、*テーブル分岐* P. A6-24 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

STRD<c><q> <Rt>, <Rt2>, [<Rn> {, #+/-<imm>}]	Offset: index==TRUE, wback==FALSE
STRD<c><q> <Rt>, <Rt2>, [<Rn>, #+/-<imm>]!	Pre-indexed: index==TRUE, wback==TRUE
STRD<c><q> <Rt>, <Rt2>, [<Rn>], #+/-<imm>	Post-indexed: index==FALSE, wback==TRUE

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rt> The first source register. For an ARM instruction, <Rt> must be even-numbered and not R14.

<Rt2> The second source register. For an ARM instruction, <Rt2> must be <R(t+1)>.

<Rn> The base register. The SP can be used. In the ARM instruction set, for offset addressing only, the PC can be used. However, use of the PC is deprecated.

+/- Is + or omitted if the immediate offset is to be added to the base register value (add == TRUE), or – if it is to be subtracted (add == FALSE). #0 and #-0 generate different instructions.

<imm> The immediate offset used to form the address. Values are multiples of 4 in the range 0-1020 for encoding T1, and any value in the range 0-255 for encoding A1. For the offset addressing syntax, <imm> can be omitted, meaning an offset of 0.

The pre-UAL syntax STR<c>D is equivalent to STRD<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemA[address,4] = R[t];
    MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

STRD<c><q> <Rt>, <Rt2>, [<Rn> {, #+/-<imm>}]	オフセット：index==TRUE、wback==FALSE
STRD<c><q> <Rt>, <Rt2>, [<Rn>, #+/-<imm>]!	プリインデクス：index==TRUE、wback==TRUE
STRD<c><q> <Rt>, <Rt2>, [<Rn>], #+/-<imm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rt> 最初のソースレジスタ。ARM 命令の場合、<Rt> には R14 以外の偶数番号のレジスタのみが使用できます。

<Rt2> 2 番目のソースレジスタ。ARM 命令の場合、<Rt2> は <R(t+1)> でなければなりません。

<Rn> ベースレジスタ。このレジスタには SP を指定できます。ARM 命令セットでは、オフセットアドレシングに対してのみ PC を指定できます。しかし、PC の使用は推奨されません。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、「+」にするか、省略します。減算する場合 (add == FALSE) は、「-」にします。#0 と #-0 では異なる命令が生成されます。

<imm> アドレスを形成するために使用されるイミディエートオフセット。値の範囲は、エンコード T1 では 4 の倍数で 0 ～ 1020、エンコード A1 では 0 ～ 255 の任意の値です。オフセットアドレシング構文では <imm> を省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の STR<c>D は、STRD<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemA[address,4] = R[t];
    MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.201 STRD (register)

Store Register Dual (register) calculates an address from a base register value and a register offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding A1 ARMv5TE*, ARMv6*, ARMv7

```
STRD<c> <Rt>,<Rt2>,[<Rn>,+/-<Rm>]{!}
STRD<c> <Rt>,<Rt2>,[<Rn>],+/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	0	W	0	Rn	Rt	(0) (0) (0) (0)	1 1 1 1	Rm

```
if Rt<0> == '1' then UNDEFINED;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if t2 == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.201 STRD（レジスタ）

STRD（デュアル レジスタストア（レジスタ））は、ベースレジスタの値とレジスタオフセットからアドレスを計算して、2 つのレジスタからそのメモリに 2 ワードをストアします。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス*P. A8-13 を参照して下さい。

エンコード A1 ARMv5TE*、ARMv6*、ARMv7

```
STRD<c> <Rt>,<Rt2>,[<Rn>,+/-<Rm>]{!}
STRD<c> <Rt>,<Rt2>,[<Rn>],+/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	0	W	0	Rn	Rt	(0) (0) (0) (0)	1 1 1 1	Rm

```
if Rt<0> == '1' then UNDEFINED;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if t2 == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STRD<c><q> <Rt>, <Rt2>, [<Rn>, +/-<Rm>]`	Offset: `index==TRUE`, `wback==FALSE`
`STRD<c><q> <Rt>, <Rt2>, [<Rn>, +/-<Rm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STRD<c><q> <Rt>, <Rt2>, [<Rn>], +/-<Rm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The first source register. This register must be even-numbered and not R14.

`<Rt2>` The second source register. This register must be `<R(t+1)>`.

`<Rn>` The base register. The SP can be used. For offset addressing only, the PC can be used. However, use of the PC is deprecated.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`).

`<Rm>` Contains the offset that is added to the value of `<Rn>` to form the address.

The pre-UAL syntax `STR<c>D` is equivalent to `STRD<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    MemA[address,4] = R[t];
    MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`STRD<c><q> <Rt>, <Rt2>, [<Rn>, +/-<Rm>]`	オフセット：`index==TRUE`、`wback==FALSE`
`STRD<c><q> <Rt>, <Rt2>, [<Rn>, +/-<Rm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`STRD<c><q> <Rt>, <Rt2>, [<Rn>], +/-<Rm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` 最初のソースレジスタ。R14 以外の偶数番号のレジスタのみが使用できます。

`<Rt2>` 2 番目のソースレジスタ。このレジスタは、`<R(t+1)>` でなければなりません。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC は、オフセットアドレシングでのみ指定できます。しかし、PC の使用は推奨されません。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合 (`add == FALSE`) は、「-」にします。

`<Rm>` アドレス生成のために `<Rn>` の値に加算されるオフセット

UAL 以前の構文の `STR<c>D` は、`STRD<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    MemA[address,4] = R[t];
    MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.202 STREX

Store Register Exclusive calculates an address from a base register value and an immediate offset, and stores a word from a register to memory if the executing processor has exclusive access to the memory addressed.

For more information about support for shared memory see *Synchronization and semaphores* on page A3-12. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
STREX<c> <Rd>,<Rt>,[<Rn>{,#<imm>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	1	0	0	Rn				Rt				Rd				imm8							

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if BadReg(d) || BadReg(t) || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
STREX<c> <Rd>,<Rt>,[<Rn>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	0	0	Rn				Rd				(1)	(1)	(1)	(1)	1	0	0	1	Rt			

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.202 STREX

STREX（排他レジスタストア）は、実行中のプロセッサがアドレス指定されたメモリに対して排他アクセス権を持っている場合に、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 ワードをストアします。

共有メモリのサポートの詳細については、*同期化とセマフォ* P. A3- 12 を参照して下さい。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
STREX<c> <Rd>,<Rt>,[<Rn>{,#<imm>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	1	0	0	Rn				Rt				Rd				imm8							

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if BadReg(d) || BadReg(t) || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
STREX<c> <Rd>,<Rt>,[<Rn>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	1	0	0	0	Rn				Rd				(1)	(1)	(1)	(1)	1	0	0	1	Rt			

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
STREX<c><q>  <Rd>, <Rt>, [<Rn> {,#<imm>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register for the returned status value. The value returned is:

- `0` if the operation updates memory
- `1` if the operation fails to update memory.

`<Rt>` The source register.

`<Rn>` The base register. The SP can be used.

`<imm>` The immediate offset added to the value of `<Rn>` to form the address. Values are multiples of 4 in the range 0-1020 for encoding T1, and 0 for encoding A1. `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + imm32;
    if ExclusiveMonitorsPass(address,4) then
        MemA[address,4] = R[t];
        R[d] = 0;
    else
        R[d] = 1;
```

Exceptions

Data Abort.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- memory is not updated
- `<Rd>` is not updated.

If `ExclusiveMonitorsPass()` returns `FALSE` and the memory address would generate a synchronous Data Abort exception if accessed, it is IMPLEMENTATION DEFINED whether the exception is generated.

If SCTLR.A and SCTLR.U are both 0, a non word-aligned memory address causes UNPREDICTABLE behavior. Otherwise, a non word-aligned memory address causes a Data Abort exception with type Alignment fault to be generated according to the following rules:

- if `ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated
- otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
STREX<c><q>  <Rd>, <Rt>, [<Rn> {,#<imm>}]
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` ステータス値が返されるデスティネーションレジスタ。返される値は次のとおりです。

- `0` 命令によってメモリが更新された場合
- `1` 命令によってメモリが更新されなかった場合

`<Rt>` ソースレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`<imm>` アドレス生成のために `<Rn>` の値に加算されるイミディエートオフセット。値は、エンコード T1 では 4 の倍数で 0 ～ 1020、エンコード A1 では 0 です。`<imm>` は省略可能で、その場合オフセットは 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + imm32;
    if ExclusiveMonitorsPass(address,4) then
        MemA[address,4] = R[t];
        R[d] = 0;
    else
        R[d] = 1;
```

例外

データアボート

アボートとアライメント

この命令の実行によって同期データアボート例外が生成される場合、次のようになります。

- メモリは更新されません。
- `<Rd>` は更新されません。

`ExclusiveMonitorsPass()` から `FALSE` が返され、アクセス時にそのメモリアドレスで同期データアボート例外が生成される可能性がある場合、この例外が生成されるかどうかは実装定義です。

SCTLR.A と SCTLR.U の両方が 0 の場合、ワードアラインドでないメモリアドレスによって、予測不能な動作が生じます。それ以外の場合、次の規則に従って、ワードアラインドでないメモリアドレスによるデータアボート例外が発生し、アライメント型のフォールトが生成されます。

- `ExclusiveMonitorsPass()` から `TRUE` が返される場合、この例外が生成されます。
- それ以外の場合、この例外が生成されるかどうかは実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.203 STREXB

Store Register Exclusive Byte derives an address from a base register value, and stores a byte from a register to memory if the executing processor has exclusive access to the memory addressed.

For more information about support for shared memory see *Synchronization and semaphores* on page A3-12. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv7

```
STREXB<c> <Rd>,<Rt>,[<Rn>]
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 1	0 0 0 1 1 0	0	Rn	Rt	(1) (1) (1) (1)	0 1 0 0	Rd

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if BadReg(d) || BadReg(t) || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

Encoding A1 ARMv6K, ARMv7

```
STREXB<c> <Rd>,<Rt>,[<Rn>]
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	1 1 1 0	0	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rt

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.203 STREXB

STREXB（排他レジスタストア バイト）は、実行中のプロセッサがアドレス指定されたメモリに対して排他アクセス権を持っている場合に、ベースレジスタの値からアドレスを計算して、レジスタからそのメモリへ 1 バイトをストアします。

共有メモリのサポートの詳細については、*同期化とセマフォ* P. A3-12 を参照して下さい。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv7

```
STREXB<c> <Rd>,<Rt>,[<Rn>]
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 1	0 0 0 1 1 0	0	Rn	Rt	(1) (1) (1) (1)	0 1 0 0	Rd

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if BadReg(d) || BadReg(t) || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

エンコード A1 ARMv6K、ARMv7

```
STREXB<c> <Rd>,<Rt>,[<Rn>]
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	1 1 1 0	0	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rt

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
STREXB<c><q>  <Rd>, <Rt>, [<Rn>]
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register for the returned status value. The value returned is:

- 0 if the operation updates memory
- 1 if the operation fails to update memory.

<Rt> The source register.

<Rn> The base register. The SP can be used.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    if ExclusiveMonitorsPass(address,1) then
        MemA[address,1] = R[t];
        R[d] = 0;
    else
        R[d] = 1;
```

Exceptions

Data Abort.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- memory is not updated
- <Rd> is not updated.

If ExclusiveMonitorsPass() returns FALSE and the memory address would generate a synchronous Data Abort exception if accessed, it is IMPLEMENTATION DEFINED whether the exception is generated.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
STREXB<c><q>  <Rd>, <Rt>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> ステータス値が返されるデスティネーションレジスタ。返される値は次のとおりです。

- 0 命令によってメモリが更新された場合
- 1 命令によってメモリが更新されなかった場合

<Rt> ソースレジスタ

<Rn> ベースレジスタ。このレジスタには SP を指定できます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    if ExclusiveMonitorsPass(address,1) then
        MemA[address,1] = R[t];
        R[d] = 0;
    else
        R[d] = 1;
```

例外

データアボート

アボート

この命令の実行によって同期データアボート例外が生成される場合、次のようになります。

- メモリは更新されません。
- <Rd> は更新されません。

ExclusiveMonitorsPass() から FALSE が返され、アクセス時にそのメモリアドレスで同期データアボート例外が生成される可能性がある場合、この例外が生成されるかどうかは実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.204 STREXD

Store Register Exclusive Doubleword derives an address from a base register value, and stores a 64-bit doubleword from two registers to memory if the executing processor has exclusive access to the memory addressed.

For more information about support for shared memory see *Synchronization and semaphores* on page A3-12. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv7

```
STREXD<c> <Rd>,<Rt>,<Rt2>,[<Rn>]
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 1	0 0 0 1 1 0	0	Rn	Rt	Rt2	0 1 1 1	Rd

```
d = UInt(Rd);  t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if BadReg(d) || BadReg(t) || BadReg(t2) || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

Encoding A1 ARMv6K, ARMv7

```
STREXD<c> <Rd>,<Rt>,<Rt2>,[<Rn>]
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	1 1 0 1	0	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rt

```
d = UInt(Rd);  t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);
if d == 15 || Rt<0> = '1' || Rt == '1110' || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.204 STREXD

STREXD（排他レジスタストア ダブルワード）は、実行中のプロセッサがアドレス指定されたメモリに対して排他アクセス権を持っている場合に、ベースレジスタの値からアドレスを計算して、2 つのレジスタからそのメモリへ 64 ビットダブルワードをストアします。

共有メモリのサポートの詳細については、*同期化とセマフォ* P. A3- 12 を参照して下さい。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード T1 ARMv7

```
STREXD<c> <Rd>,<Rt>,<Rt2>,[<Rn>]
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 1	0 0 0 1 1 0	0	Rn	Rt	Rt2	0 1 1 1	Rd

```
d = UInt(Rd);  t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if BadReg(d) || BadReg(t) || BadReg(t2) || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

エンコード A1 ARMv6K、ARMv7

```
STREXD<c> <Rd>,<Rt>,<Rt2>,[<Rn>]
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	1 1 0 1	0	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rt

```
d = UInt(Rd);  t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);
if d == 15 || Rt<0> = '1' || Rt == '1110' || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
STREXD<c><q>  <Rd>, <Rt>, <Rt2>, [<Rn>]
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register for the returned status value. The value returned is:

- 0 if the operation updates memory
- 1 if the operation fails to update memory.

<Rt> The first source register.

<Rt2> The second source register.

<Rn> The base register. The SP can be used.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian() then R[t]:R[t2] else R[t2]:R[t];
    if ExclusiveMonitorsPass(address,8) then
        MemA[address,8] = value;  R[d] = 0;
    else
        R[d] = 1;
```

Exceptions

Data Abort.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- memory is not updated
- <Rd> is not updated.

If `ExclusiveMonitorsPass()` returns `FALSE` and the memory address would generate a synchronous Data Abort exception if accessed, it is IMPLEMENTATION DEFINED whether the exception is generated.

If SCTLR.A and SCTLR.U are both 0, a non doubleword-aligned memory address causes UNPREDICTABLE behavior. Otherwise, a non doubleword-aligned memory address causes a Data Abort exception with type Alignment fault to be generated according to the following rules:

- if `ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated
- otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
STREXD<c><q>  <Rd>, <Rt>, <Rt2>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> ステータス値が返されるデスティネーションレジスタ。返される値は次のとおりです。

- 0 命令によってメモリが更新された場合
- 1 命令によってメモリが更新されなかった場合

<Rt> 最初のソースレジスタ

<Rt2> 2 番目のソースレジスタ

<Rn> ベースレジスタ。このレジスタには SP を指定できます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian() then R[t]:R[t2] else R[t2]:R[t];
    if ExclusiveMonitorsPass(address,8) then
        MemA[address,8] = value;  R[d] = 0;
    else
        R[d] = 1;
```

例外

データアボート

アボートとアライメント

この命令の実行によって同期データアボート例外が生成される場合、次のようになります。

- メモリは更新されません。
- <Rd> は更新されません。

`ExclusiveMonitorsPass()` から `FALSE` が返され、アクセス時にそのメモリアドレスで同期データアボート例外が生成される可能性がある場合、この例外が生成されるかどうかは実装定義です。

SCTLR.A と SCTLR.U の両方が 0 の場合、ダブルワードアラインドでないメモリアドレスによって、予測不能な動作が生じます。それ以外の場合、次の規則に従って、ダブルワードアラインドでないメモリアドレスによってデータアボート例外が発生し、アライメント型のフォールトが生成されます。

- `ExclusiveMonitorsPass()` から `TRUE` が返される場合、この例外が生成されます。
- それ以外の場合、この例外が生成されるかどうかは実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.205 STREXH

Store Register Exclusive Halfword derives an address from a base register value, and stores a halfword from a register to memory if the executing processor has exclusive access to the memory addressed.

For more information about support for shared memory see *Synchronization and semaphores* on page A3-12. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv7

```
STREXH<c> <Rd>,<Rt>,[<Rn>]
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 1	0 0 0 1 1 0	0	Rn	Rt	(1) (1) (1) (1)	0 1 0 1	Rd

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if BadReg(d) || BadReg(t) || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

Encoding A1 ARMv6K, ARMv7

```
STREXH<c> <Rd>,<Rt>,[<Rn>]
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	1 1 1 1	0	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rt

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.205 STREXH

STREXH（排他レジスタストア ハーフワード）は、実行中のプロセッサがアドレス指定されたメモリに対して排他アクセス権を持っている場合に、ベースレジスタの値からアドレスを計算して、レジスタからそのメモリへ 1 ハーフワードをストアします。

共有メモリのサポートの詳細については、*同期化とセマフォ* P. A3-12 を参照して下さい。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

エンコード T1 ARMv7

```
STREXH<c> <Rd>,<Rt>,[<Rn>]
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 1	0 0 0 1 1 0	0	Rn	Rt	(1) (1) (1) (1)	0 1 0 1	Rd

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if BadReg(d) || BadReg(t) || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

エンコード A1 ARMv6K、ARMv7

```
STREXH<c> <Rd>,<Rt>,[<Rn>]
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	1 1 1 1	0	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rt

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
STREXH<c><q>  <Rd>, <Rt>, [<Rn>]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register for the returned status value. The value returned is:

- `0` if the operation updates memory
- `1` if the operation fails to update memory.

`<Rt>` The source register.

`<Rn>` The base register. The SP can be used.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    if ExclusiveMonitorsPass(address,2) then
        MemA[address,2] = R[t];
        R[d] = 0;
    else
        R[d] = 1;
```

Exceptions

Data Abort.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- memory is not updated
- `<Rd>` is not updated.

If `ExclusiveMonitorsPass()` returns `FALSE` and the memory address would generate a synchronous Data Abort exception if accessed, it is IMPLEMENTATION DEFINED whether the exception is generated.

If SCTLR.A and SCTLR.U are both 0, a non halfword-aligned memory address causes UNPREDICTABLE behavior. Otherwise, a non halfword-aligned memory address causes a Data Abort exception with type Alignment fault to be generated according to the following rules:

- if `ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated
- otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
STREXH<c><q>  <Rd>, <Rt>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` ステータス値が返されるデスティネーションレジスタ。返される値は次のとおりです。

- `0` 命令によってメモリが更新された場合
- `1` 命令によってメモリが更新されなかった場合

`<Rt>` ソースレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n];
    if ExclusiveMonitorsPass(address,2) then
        MemA[address,2] = R[t];
        R[d] = 0;
    else
        R[d] = 1;
```

例外

データアボート

アボートとアライメント

この命令の実行によって同期データアボート例外が生成される場合、次のようになります。

- メモリは更新されません。
- `<Rd>` は更新されません。

`ExclusiveMonitorsPass()` から `FALSE` が返され、アクセス時にそのメモリアドレスで同期データアボート例外が生成される可能性がある場合、この例外が生成されるかどうかは実装定義です。

SCTLR.A と SCTLR.U の両方が 0 の場合、ハーフワードアラインドでないメモリアドレスによって、予測不能な動作が生じます。それ以外の場合、次の規則に従って、ハーフワードアラインドでないメモリアドレスによってデータアボート例外が発生し、アライメント型のフォールトが生成されます。

- `ExclusiveMonitorsPass()` から `TRUE` が返される場合、この例外が生成されます。
- それ以外の場合、この例外が生成されるかどうかは実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.206 STRH (immediate, Thumb)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
STRH<c> <Rt>,[<Rn>{,#<imm>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 ARMv6T2, ARMv7

```
STRH<c>.W <Rt>,[<Rn>{,#<imm12>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	1	0	Rn				Rt				imm12											

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if BadReg(t) then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

```
STRH<c> <Rt>,[<Rn>,#-<imm8>]
STRH<c> <Rt>,[<Rn>],#+/-<imm8>
STRH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	0	Rn				Rt				1	P	U	W	imm8							

```
if P == '1' && U == '1' && W == '0' then SEE STRHT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.206 STRH（イミディエート、Thumb）

STRH（レジスタストア ハーフワード（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 ハーフワードをストアします。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス*P. A8- 13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
STRH<c> <Rt>,[<Rn>{,#<imm>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T2 ARMv6T2、ARMv7

```
STRH<c>.W <Rt>,[<Rn>{,#<imm12>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	1	0	Rn				Rt				imm12											

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if BadReg(t) then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

```
STRH<c> <Rt>,[<Rn>,#-<imm8>]
STRH<c> <Rt>,[<Rn>],#+/-<imm8>
STRH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	0	Rn				Rt				1	P	U	W	imm8							

```
if P == '1' && U == '1' && W == '0' then SEE STRHT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STRH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`STRH<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STRH<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register.

`<Rn>` The base register. The SP can be used.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. Values are:

Encoding T1	multiples of 2 in the range 0-62
Encoding T2	any value in the range 0-4095
Encoding T3	any value in the range 0-255.

For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `STR<c>H` is equivalent to `STRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if UnalignedSupport() || address<0> == '0' then
        MemU[address,2] = R[t]<15:0>;
    else // Can only occur before ARMv7
        MemU[address,2] = bits(16) UNKNOWN;
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`STRH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`STRH<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`STRH<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` ソースレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。値は次のとおりです。

エンコード T1	2 の倍数で 0 ～ 62
エンコード T2	0 ～ 4095 の任意の値
エンコード T3	0 ～ 255 の任意の値

オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の `STR<c>H` は、`STRH<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if UnalignedSupport() || address<0> == '0' then
        MemU[address,2] = R[t]<15:0>;
    else // Can only occur before ARMv7
        MemU[address,2] = bits(16) UNKNOWN;
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.207 STRH (immediate, ARM)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STRH<c> <Rt>,[<Rn>{,#+/-<imm8>}]
STRH<c> <Rt>,[<Rn>],#+/-<imm8>
STRH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	1	W	0	Rn	Rt	imm4H	1 0 1 1	imm4L

```
if P == '0' && W == '1' then SEE STRHT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.207 STRH（イミディエート、ARM）

STRH（レジスタストア ハーフワード（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 ハーフワードをストアします。オフセットアドレシング、ポストインデクス アドレシング、プリインデクス アドレシングを使用できます。メモリアクセスの詳細については、*メモリアクセス*P. A8- 13 を参照して下さい。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STRH<c> <Rt>,[<Rn>{,#+/-<imm8>}]
STRH<c> <Rt>,[<Rn>],#+/-<imm8>
STRH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	1	W	0	Rn	Rt	imm4H	1 0 1 1	imm4L

```
if P == '0' && W == '1' then SEE STRHT;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STRH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`STRH<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STRH<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register.

`<Rn>` The base register. The SP can be used. For offset addressing only, the PC can be used. However, use of the PC is deprecated.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. Values are 0-255. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `STR<c>H` is equivalent to `STRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if UnalignedSupport() || address<0> == '0' then
        MemU[address,2] = R[t]<15:0>;
    else // Can only occur before ARMv7
        MemU[address,2] = bits(16) UNKNOWN;
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`STRH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	オフセット：`index==TRUE`、`wback==FALSE`
`STRH<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	プリインデクス：`index==TRUE`、`wback==TRUE`
`STRH<c><q> <Rt>, [<Rn>], #+/-<imm>`	ポストインデクス：`index==FALSE`、`wback==TRUE`

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` ソースレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。PC は、オフセットアドレシングでのみ指定できます。しかし、PC の使用は推奨されません。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（`add == FALSE`）は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。値は 0 ～ 255 です。オフセットアドレシング構文では `<imm>` を省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の `STR<c>H` は、`STRH<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if UnalignedSupport() || address<0> == '0' then
        MemU[address,2] = R[t]<15:0>;
    else // Can only occur before ARMv7
        MemU[address,2] = bits(16) UNKNOWN;
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.208 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
STRH<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	0 0 1	Rm	Rn	Rt

```
if CurrentInstrSet() == InstrSet_ThumbEE then SEE "Modified operation in ThumbEE";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
STRH<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12	11 10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8 7 6	5 4 3 2 1 0
1 1 1 1 1	0 0	0	0	0 1	0	Rn	Rt	0	0 0 0 0 0	imm2 Rm

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(t) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STRH<c> <Rt>,[<Rn>,+/-<Rm>]{!}
STRH<c> <Rt>,[<Rn>],+/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	0	W	0	Rn	Rt	(0) (0) (0) (0)	1 0 1 1	Rm

```
if P == '0' && W == '1' then SEE STRHT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

Modified operation in ThumbEE See *STRH (register)* on page A9-13

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.208 STRH（レジスタ）

STRH（レジスタストア ハーフワード（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、レジスタからそのメモリに 1 ハーフワードをストアします。オフセットレジスタの値は、左に 0、1、2、3 ビットシフトできます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
STRH<c> <Rt>,[<Rn>,<Rm>]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	0 0 1	Rm	Rn	Rt

```
if CurrentInstrSet() == InstrSet_ThumbEE then SEE "Modified operation in ThumbEE";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
STRH<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15 14 13 12	11 10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8 7 6	5 4 3 2 1 0
1 1 1 1 1	0 0	0	0	0 1	0	Rn	Rt	0	0 0 0 0 0	imm2 Rm

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(t) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STRH<c> <Rt>,[<Rn>,+/-<Rm>]{!}
STRH<c> <Rt>,[<Rn>],+/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	P	U	0	W	0	Rn	Rt	(0) (0) (0) (0)	1 0 1 1	Rm

```
if P == '0' && W == '1' then SEE STRHT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

ThumbEE で変更された動作 *STRH（レジスタ）* P. A9- 13 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

STRH<c><q> <Rt>, [<Rn>, +/-<Rm>{, LSL #<imm>}]	Offset: index==TRUE, wback==FALSE
STRH<c><q> <Rt>, [<Rn>, +/-<Rm>]!	Pre-indexed: index==TRUE, wback==TRUE
STRH<c><q> <Rt>, [<Rn>], +/-<Rm>	Post-indexed: index==FALSE, wback==TRUE

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rt> The source register.

<Rn> The base register. The SP can be used. In the ARM instruction set, for offset addressing only, the PC can be used. However, use of the PC is deprecated.

+/- Is + or omitted if the optionally shifted value of <Rm> is to be added to the base register value (add == TRUE), or – if it is to be subtracted (permitted in ARM code only, add == FALSE).

<Rm> Contains the offset that is optionally left shifted and added to the value of <Rn> to form the address.

<imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. Only encoding T2 is permitted, and <imm> is encoded in imm2.

If absent, no shift is specified and all encodings are permitted. In encoding T2, imm2 is encoded as 0b00.

The pre-UAL syntax STR<c>H is equivalent to STRH<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    if UnalignedSupport() || address<0> == '0' then
        MemU[address,2] = R[t]<15:0>;
    else // Can only occur before ARMv7
        MemU[address,2] = bits(16) UNKNOWN;
    if wback then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

STRH<c><q> <Rt>, [<Rn>, +/-<Rm>{, LSL #<imm>}]	オフセット：index==TRUE、wback==FALSE
STRH<c><q> <Rt>, [<Rn>, +/-<Rm>]!	プリインデクス：index==TRUE、wback==TRUE
STRH<c><q> <Rt>, [<Rn>], +/-<Rm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rt> ソースレジスタ

<Rn> ベースレジスタ。このレジスタには SP を指定できます。ARM 命令セットでは、オフセットアドレシングに対してのみ PC を指定できます。しかし、PC の使用は推奨されません。

+/- オプションとしてシフトされる <Rm> の値をベースレジスタの値に加算する場合 (add == TRUE) は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、add == FALSE）は、「-」にします。

<Rm> アドレス生成のためにオプションとして左にシフトされて、<Rn> の値に加算されるオフセット

<imm> 指定した場合、左シフト量 (1 ～ 3) が <Rm> から読み出された値に適用されます。エンコード T2 のみ許可され、<imm> は imm2 にエンコードされます。

省略した場合、シフトは指定されず、すべてのエンコードが許可されます。エンコード T2 では、imm2 は 0b00 としてエンコードされます。

UAL 以前の構文の STR<c>H は、STRH<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    if UnalignedSupport() || address<0> == '0' then
        MemU[address,2] = R[t]<15:0>;
    else // Can only occur before ARMv7
        MemU[address,2] = bits(16) UNKNOWN;
    if wback then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.209 STRHT

Store Register Halfword Unprivileged and stores a halfword from a register to memory. For information about memory accesses see *Memory accesses* on page A8-13.

The memory access is restricted as if the processor were running in User mode. (This makes no difference if the processor is actually running in User mode.)

The Thumb instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The ARM instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

Encoding T1 ARMv6T2, ARMv7

```
STRHT<c> <Rt>,[<Rn>,#<imm8>]
```

15 14 13 12 11	10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 1	0 0	0	0	0 1	0	Rn	Rt	1	1 1 0	imm8

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

Encoding A1 ARMv6T2, ARMv7

```
STRHT<c> <Rt>, [<Rn>] {, #+/-<imm8>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	0	U	1	1	0	Rn	Rt	imm4H	1 0 1 1	imm4L

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

Encoding A2 ARMv6T2, ARMv7

```
STRHT<c> <Rt>, [<Rn>], +/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	0	U	0	1	0	Rn	Rt	(0) (0) (0) (0)	1 0 1 1	Rm

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.209 STRHT

STRHT（非特権レジスタストア ハーフワード）は、レジスタからメモリに 1 ハーフワードをストアします。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

プロセッサがユーザモードで実行されている場合と同様にメモリアクセスが制限されます（実際にプロセッサがユーザモードで実行されている場合は、何も変わりません）。

Thumb 命令では、オフセット アドレシングモードを使用します。このモードでは、メモリアクセスに使用されるアドレスを、ベースレジスタの値とイミディエートオフセットから計算し、ベースレジスタの値は変更しません。

ARM 命令では、ポストインデクス アドレシングモードを使用します。このモードでは、ベースレジスタの値をメモリアクセス用のアドレスとして使用し、ベースレジスタの値とオフセットから新しいアドレスを計算して、ベースレジスタに書き戻します。オフセットは、イミディエート値またはレジスタの値です。

エンコード T1 ARMv6T2、ARMv7

```
STRHT<c> <Rt>,[<Rn>,#<imm8>]
```

15 14 13 12 11	10 9	8	7	6 5	4	3 2 1 0	15 14 13 12	11	10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 1	0 0	0	0	0 1	0	Rn	Rt	1	1 1 0	imm8

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

エンコード A1 ARMv6T2、ARMv7

```
STRHT<c> <Rt>, [<Rn>] {, #+/-<imm8>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	0	U	1	1	0	Rn	Rt	imm4H	1 0 1 1	imm4L

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

エンコード A2 ARMv6T2、ARMv7

```
STRHT<c> <Rt>, [<Rn>], +/-<Rm>
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0	0	U	0	1	0	Rn	Rt	(0) (0) (0) (0)	1 0 1 1	Rm

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STRHT<c><q> <Rt>, [<Rn> {, #<imm>}]`	Offset: Thumb only
`STRHT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	Post-indexed: ARM only
`STRHT<c><q> <Rt>, [<Rn>], +/-<Rm>`	Post-indexed: ARM only

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register.

`<Rn>` The base register. The SP can be used.

`+/-` Is + or omitted if `<imm>` or the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, `add == FALSE`).

`<imm>` The immediate offset applied to the value of `<Rn>`. Any value in the range 0-255 is permitted. `<imm>` can be omitted, meaning an offset of 0.

`<Rm>` Contains the offset that is applied to the value of `<Rn>` to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    if UnalignedSupport() || address<0> == '0' then
        MemU_unpriv[address,2] = R[t]<15:0>;
    else // Can only occur before ARMv7
        MemU_unpriv[address,2] = bits(16) UNKNOWN;
    if postindex then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`STRHT<c><q> <Rt>, [<Rn> {, #<imm>}]`	オフセット：Thumb のみ
`STRHT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	ポストインデクス：ARM のみ
`STRHT<c><q> <Rt>, [<Rn>], +/-<Rm>`	ポストインデクス：ARM のみ

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>` ソースレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`+/-` `<imm>` またはオプションとしてシフトされる `<Rm>` の値をベースレジスタの値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、`add == FALSE`）は、「-」にします。

`<imm>` `<Rn>` の値に適用されるイミディエートオフセット。0 ～ 255 の任意の値が許可されます。`<imm>` は省略可能で、その場合オフセットは 0 と見なされます。

`<Rm>` アドレス生成のために `<Rn>` の値に適用されるオフセットを格納します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    if UnalignedSupport() || address<0> == '0' then
        MemU_unpriv[address,2] = R[t]<15:0>;
    else // Can only occur before ARMv7
        MemU_unpriv[address,2] = bits(16) UNKNOWN;
    if postindex then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.210 STRT

Store Register Unprivileged and stores a word from a register to memory. For information about memory accesses see *Memory accesses* on page A8-13.

The memory access is restricted as if the processor were running in User mode. (This makes no difference if the processor is actually running in User mode.)

The Thumb instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The ARM instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

Encoding T1 ARMv6T2, ARMv7

```
STRT<c> <Rt>,[<Rn>,#<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	0	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STRT<c> <Rt>, [<Rn>] {, +/-<imm12>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	0	0	U	0	1	0	Rn				Rt				imm12											

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if n == 15 || n == t then UNPREDICTABLE;
```

Encoding A2 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STRT<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	U	0	1	0	Rn				Rt				imm5				type		0	Rm				

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(type, imm5);
if n == 15 || n == t || m == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && m == n then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.210 STRT

STRT（非特権レジスタストア）は、レジスタからメモリに 1 ワードをストアします。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

プロセッサがユーザモードで実行されている場合と同様にメモリアクセスが制限されます（実際にプロセッサがユーザモードで実行されている場合は、何も変わりません）。

Thumb 命令では、オフセット アドレシングモードを使用します。このモードでは、メモリアクセスに使用されるアドレスを、ベースレジスタの値とイミディエートオフセットから計算し、ベースレジスタの値は変更しません。

ARM 命令では、ポストインデクス アドレシングモードを使用します。このモードでは、ベースレジスタの値をメモリアクセス用のアドレスとして使用し、ベースレジスタの値とオフセットから新しいアドレスを計算して、ベースレジスタに書き戻します。オフセットは、イミディエート値またはオプションとしてシフトされるレジスタの値です。

エンコード T1 ARMv6T2、ARMv7

```
STRT<c> <Rt>,[<Rn>,#<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	0	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STRT<c> <Rt>, [<Rn>] {, +/-<imm12>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	0	0	U	0	1	0	Rn				Rt				imm12											

```
t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if n == 15 || n == t then UNPREDICTABLE;
```

エンコード A2 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STRT<c> <Rt>,[<Rn>],+/-<Rm>{, <shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	U	0	1	0	Rn				Rt				imm5				type		0	Rm				

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(type, imm5);
if n == 15 || n == t || m == 15 then UNPREDICTABLE;
if ArchVersion() < 6 && m == n then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`STRT<c><q> <Rt>, [<Rn> {, #<imm>}]`	Offset: Thumb only
`STRT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	Post-indexed: ARM only
`STRT<c><q> <Rt>, [<Rn>], +/-<Rm> {, <shift>}`	Post-indexed: ARM only

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register. In the ARM instruction set, the PC can be used. However, use of the PC is deprecated.

`<Rn>` The base register. The SP can be used.

`+/-` Is + or omitted if `<imm>` or the optionally shifted value of `<Rm>` is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (permitted in ARM code only, `add == FALSE`).

`<imm>` The immediate offset applied to the value of `<Rn>`. Values are 0-255 for encoding T1, and 0-4095 for encoding A1. `<imm>` can be omitted, meaning an offset of 0.

`<Rm>` Contains the offset that is optionally shifted and added to the value of `<Rn>` to form the address.

`<shift>` The shift to apply to the value read from `<Rm>`. If omitted, no shift is applied. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

The pre-UAL syntax `STR<c>T` is equivalent to `STRT<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then Shift(R[m], shift_t, shift_n, APSR.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    if t == 15 then  // Only possible for encodings A1 and A2
        data = PCStoreValue();
    else
        data = R[t];
    if UnalignedSupport() || address<1:0> == '00' || CurrentInstrSet() == InstrSet_ARM then
        MemU_unpriv[address,4] = data;
    else // Can only occur before ARMv7
        MemU_unpriv[address,4] = bits(32) UNKNOWN;
    if postindex then R[n] = offset_addr;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`STRT<c><q> <Rt>, [<Rn> {, #<imm>}]`	オフセット：Thumb のみ
`STRT<c><q> <Rt>, [<Rn>] {, #+/-<imm>}`	ポストインデクス：ARM のみ
`STRT<c><q> <Rt>, [<Rn>], +/-<Rm> {, <shift>}`	ポストインデクス：ARM のみ

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>` ソースレジスタ。ARM 命令セットでは、PC を指定できます。しかし、PC の使用は推奨されません。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。

`+/-` `<imm>` またはオプションとしてシフトされる `<Rm>` の値をベースレジスタの値に加算する場合（`add == TRUE`）は、「+」にするか、省略します。減算する場合（ARM コードでのみ許可、`add == FALSE`）は、「-」にします。

`<imm>` `<Rn>` の値に適用されるイミディエートオフセット。値の範囲は、エンコード T1 では 0 ～ 255、エンコード A1 では 0 ～ 4095 です。`<imm>` は省略可能で、その場合オフセットは 0 と見なされます。

`<Rm>` アドレス生成のためにオプションとしてシフトされ、`<Rn>` の値に加算されるオフセット

`<shift>` `<Rm>` から読み出された値に適用されるシフト。省略した場合、シフトは適用されません。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8- 10 を参照して下さい。

UAL 以前の構文の `STR<c>T` は、`STRT<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    offset = if register_form then Shift(R[m], shift_t, shift_n, APSR.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    if t == 15 then  // Only possible for encodings A1 and A2
        data = PCStoreValue();
    else
        data = R[t];
    if UnalignedSupport() || address<1:0> == '00' || CurrentInstrSet() == InstrSet_ARM then
        MemU_unpriv[address,4] = data;
    else // Can only occur before ARMv7
        MemU_unpriv[address,4] = bits(32) UNKNOWN;
    if postindex then R[n] = offset_addr;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.211 SUB (immediate, Thumb)

This instruction subtracts an immediate value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
SUBS <Rd>,<Rn>,#<imm3>          Outside IT block.
SUB<c> <Rd>,<Rn>,#<imm3>        Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	imm3			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
SUBS <Rdn>,#<imm8>              Outside IT block.
SUB<c> <Rdn>,#<imm8>            Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	Rdn				imm8						

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Encoding T3 ARMv6T2, ARMv7

```
SUB{S}<c>.W <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	S		Rn			0	imm3				Rd					imm8					

```
if Rd == '1111' && setflags then SEE CMP (immediate);
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || n == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv6T2, ARMv7

```
SUBW<c> <Rd>,<Rn>,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	0	1	0		Rn			0	imm3				Rd					imm8					

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if BadReg(d) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.211 SUB（イミディエート、Thumb）

SUB（減算（イミディエート、Thumb））は、レジスタの値からイミディエート値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
SUBS <Rd>,<Rn>,#<imm3>          IT ブロック外部
SUB<c> <Rd>,<Rn>,#<imm3>        IT ブロック内部
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	imm3			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
SUBS <Rdn>,#<imm8>              IT ブロック外部
SUB<c> <Rdn>,#<imm8>            IT ブロック内部
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	Rdn				imm8						

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

エンコード T3 ARMv6T2、ARMv7

```
SUB{S}<c>.W <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	S		Rn			0	imm3				Rd					imm8					

```
if Rd == '1111' && setflags then SEE CMP (immediate);
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || n == 15 then UNPREDICTABLE;
```

エンコード T4 ARMv6T2、ARMv7

```
SUBW<c> <Rd>,<Rn>,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	0	1	0		Rn			0	imm3				Rd					imm8					

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if BadReg(d) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`SUB{S}<c><q> {<Rd>,} <Rn>, #<const>`	All encodings permitted
`SUBW<c><q> {<Rd>,} <Rn>, #<const>`	Only encoding T4 permitted

where:

`S` If `S` is present, the instruction updates the flags. Otherwise, the flags are not updated.

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rn>` The first operand register. If the SP is specified for `<Rn>`, see *SUB (SP minus immediate)* on page A8-426. If the PC is specified for `<Rn>`, see *ADR* on page A8-32.

`<const>` The immediate value to be subtracted from the value obtained from `<Rn>`. The range of values is 0-7 for encoding T1, 0-255 for encoding T2 and 0-4095 for encoding T4. See *Modified immediate constants in Thumb instructions* on page A6-17 for the range of values for encoding T3.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the `SUBW` syntax). Encoding T1 is preferred to encoding T2 if `<Rd>` is specified and encoding T2 is preferred to encoding T1 if `<Rd>` is omitted.

The pre-UAL syntax `SUB<c>S` is equivalent to `SUBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`SUB{S}<c><q> {<Rd>,} <Rn>, #<const>`	すべてのエンコードが許可される
`SUBW<c><q> {<Rd>,} <Rn>, #<const>`	エンコード T4 のみが許可される

各項目の説明については以下を参照して下さい。

`S` `S` を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rn>` 最初のオペランドレジスタ。`<Rn>` に SP を指定した場合については、*SUB（SP－イミディエート）* P. A8-426 を参照して下さい。`<Rn>` に PC を指定した場合については、*ADR* P. A8-32 を参照して下さい。

`<const>` `<Rn>` から取得された値から減算されるイミディエート値。値の範囲は、エンコード T1 では 0 ～ 7、エンコード T2 では 0 ～ 255、エンコード T4 では 0 ～ 4095 です。エンコード T3 での値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 を参照して下さい。

同じ長さの複数のエンコードが使用可能な場合は、エンコード T4 よりもエンコード T3 の方が適しています（エンコード T4 が必要な場合は、`SUBW` 構文を使用して下さい）。`<Rd>` を指定する場合は、エンコード T2 よりもエンコード T1 の方が適しています。`<Rd>` を省略する場合は、エンコード T1 よりもエンコード T2 の方が適しています。

UAL 以前の構文の `SUB<c>S` は、`SUBS<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.212 SUB (immediate, ARM)

This instruction subtracts an immediate value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
SUB{S}<c> <Rd>,<Rn>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 0 1 0	S	Rn	Rd	imm12

```
if Rn == '1111' && S == '0' then SEE ADR;
if Rn == '1101' then SEE SUB (SP minus immediate);
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.212 SUB（イミディエート、ARM）

SUB（減算（イミディエート、ARM））は、レジスタの値からイミディエート値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
SUB{S}<c> <Rd>,<Rn>,#<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 0 1 0	S	Rn	Rd	imm12

```
if Rn == '1111' && S == '0' then SEE ADR;
if Rn == '1101' then SEE SUB (SP minus immediate);
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SUB{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register. If the SP is specified for <Rn>, see *SUB (SP minus immediate)* on page A8-426. If the PC is specified for <Rn>, see *ADR* on page A8-32.
<const>	The immediate value to be subtracted from the value obtained from <Rn>. See *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

The pre-UAL syntax SUB<c>S is equivalent to SUBS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SUB{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ。<Rn> に SP を指定した場合については、*SUB（SP－イミディエート）* P. A8-426 を参照して下さい。<Rn> に PC を指定した場合については、*ADR* P. A8-32 を参照して下さい。
<const>	<Rn> から取得された値から減算されるイミディエート値。値の範囲については、*ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

UAL 以前の構文の SUB<c>S は、SUBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.213 SUB (register)

This instruction subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
SUBS <Rd>,<Rn>,<Rm>                    Outside IT block.
SUB<c> <Rd>,<Rn>,<Rm>                  Inside IT block.
```

15 14 13	12 11	10	9	8 7 6	5 4 3	2 1 0
0 0 0	1 1	0	1	Rm	Rn	Rd

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
SUB{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	1 1 0 1	S	Rn	(0)	imm3	Rd	imm2	type	Rm

```
if Rd == '1111' && S == '1' then SEE CMP (register);
if Rn == '1101' then SEE SUB (SP minus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || n == 15 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
SUB{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
cond	0 0	0	0 0 1 0	S	Rn	Rd	imm5	type	0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
if Rn == '1101' then SEE SUB (SP minus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.213 SUB（レジスタ）

SUB（減算（レジスタ））は、レジスタの値から、オプションとしてシフトされたレジスタの値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
SUBS <Rd>,<Rn>,<Rm>                    IT ブロック外部
SUB<c> <Rd>,<Rn>,<Rm>                  IT ブロック内部
```

15 14 13	12 11	10	9	8 7 6	5 4 3	2 1 0
0 0 0	1 1	0	1	Rm	Rn	Rd

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
SUB{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	1 1 0 1	S	Rn	(0)	imm3	Rd	imm2	type	Rm

```
if Rd == '1111' && S == '1' then SEE CMP (register);
if Rn == '1101' then SEE SUB (SP minus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || n == 15 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
SUB{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
cond	0 0	0	0 0 1 0	S	Rn	Rd	imm5	type	0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
if Rn == '1101' then SEE SUB (SP minus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SUB{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<Rd>	The destination register.
<Rn>	The first operand register. If the SP is specified for <Rn>, see *SUB (SP minus register)* on page A8-428.
<Rm>	The register that is optionally shifted and used as the second operand.
<shift>	The shift to apply to the value read from <Rm>. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

The pre-UAL syntax SUB<c>S is equivalent to SUBS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SUB{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<Rd>	デスティネーションレジスタ
<Rn>	最初のオペランドレジスタ。<Rn> に SP を指定した場合については、*SUB（SP − レジスタ）* P. A8-428 を参照して下さい。
<Rm>	オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ
<shift>	<Rm> から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

UAL 以前の構文の SUB<c>S は、SUBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.214 SUB (register-shifted register)

This instruction subtracts a register-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
SUB{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 0 1 0	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.214 SUB（レジスタによりシフトしたレジスタ）

SUB（減算（レジスタによりシフトしたレジスタ））は、レジスタの値からレジスタによりシフトしたレジスタの値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
SUB{S}<c> <Rd>,<Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	0 0 1 0	S	Rn	Rd	Rs	0	type	1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SUB{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register.

<Rn> — The first operand register.

<Rm> — The register that is shifted and used as the second operand.

<type> — The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR — Arithmetic shift right, encoded as type = 0b10
- LSL — Logical shift left, encoded as type = 0b00
- LSR — Logical shift right, encoded as type = 0b01
- ROR — Rotate right, encoded as type = 0b11.

<Rs> — The register whose bottom byte contains the amount to shift by.

The pre-UAL syntax SUB<c>S is equivalent to SUBS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SUB{S}<c><q>  {<Rd>,} <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ

<Rn> — 最初のオペランドレジスタ

<Rm> — シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> — <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR — 算術右シフト。type = 0b10 としてエンコードされます。
- LSL — 論理左シフト。type = 0b00 としてエンコードされます。
- LSR — 論理右シフト。type = 0b01 としてエンコードされます。
- ROR — 右ローテート。type = 0b11 としてエンコードされます。

<Rs> — 最下位バイトにシフトのビット数を含むレジスタ

UAL 以前の構文の SUB<c>S は、SUBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.215 SUB (SP minus immediate)

This instruction subtracts an immediate value from the SP value, and writes the result to the destination register.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
SUB<c> SP,SP,#<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	1	imm7						

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

Encoding T2 ARMv6T2, ARMv7

```
SUB{S}<c>.W <Rd>,SP,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	S	1	1	0	1	0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMP (immediate);
d = UInt(Rd);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 15 then UNPREDICTABLE;
```

Encoding T3 ARMv6T2, ARMv7

```
SUBW<c> <Rd>,SP,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	0	1	0	1	1	0	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
SUB{S}<c> <Rd>,SP,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	0	1	0	S	1	1	0	1	Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.215 SUB（SP – イミディエート）

SUB（減算（SP – イミディエート））は、SP の値からイミディエート値を減算して、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
SUB<c> SP,SP,#<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	1	imm7						

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

エンコード T2 ARMv6T2、ARMv7

```
SUB{S}<c>.W <Rd>,SP,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	S	1	1	0	1	0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMP (immediate);
d = UInt(Rd);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 15 then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

```
SUBW<c> <Rd>,SP,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	0	1	0	1	1	0	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
SUB{S}<c> <Rd>,SP,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	0	1	0	S	1	1	0	1	Rd				imm12											

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`SUB{S}<c><q> {<Rd>,} SP, #<const>`	All encodings permitted
`SUBW<c><q> {<Rd>,} SP, #<const>`	Only encoding T3 permitted

where:

`S` If `S` is present, the instruction updates the flags. Otherwise, the flags are not updated.

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register. If omitted, `<Rd>` is SP.

`<const>` The immediate value to be subtracted from the value obtained from SP. Values are multiples of 4 in the range 0-508 for encoding T1 and any value in the range 0-4095 for encoding T3. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values for encodings T2 and A1.

When both 32-bit encodings are available for an instruction, encoding T2 is preferred to encoding T3 (if encoding T3 is required, use the `SUBW` syntax).

The pre-UAL syntax `SUB<c>S` is equivalent to `SUBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, NOT(imm32), '1');
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`SUB{S}<c><q> {<Rd>,} SP, #<const>`	すべてのエンコードが許可される
`SUBW<c><q> {<Rd>,} SP, #<const>`	エンコード T3 のみ許可される

各項目の説明については以下を参照して下さい。

`S` `S` を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ。省略した場合、`<Rd>` として SP が使用されます。

`<const>` SP から取得された値から減算されるイミディエート値。値の範囲は、エンコード T1 では 4 の倍数で 0 〜 508、エンコード T3 では 0 〜 4095 の任意の値です。エンコード T2 および A1 での値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

両方の 32 ビットエンコードが使用可能な場合は、エンコード T3 よりもエンコード T2 の方が適しています（エンコード T3 が必要な場合は、`SUBW` 構文を使用して下さい）。

UAL 以前の構文の `SUB<c>S` は、`SUBS<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, NOT(imm32), '1');
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.216 SUB (SP minus register)

This instruction subtracts an optionally-shifted register value from the SP value, and writes the result to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
SUB{S}<c> <Rd>,SP,<Rm>{,<shift>}
```

15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0	1	1 1 0 1	S	1 1 0 1	(0)	imm3	Rd	imm2	type	Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 && (shift_t != SRType_LSL || shift_n > 3) then UNPREDICTABLE;
if d == 15 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
SUB{S}<c> <Rd>,SP,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 0	0	0 0 1 0	S	1 1 0 1	Rd	imm5	type		0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.216 SUB（SP − レジスタ）

SUB（減算（SP − レジスタ））は、SP の値から、オプションとしてシフトされたレジスタの値を減算して、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
SUB{S}<c> <Rd>,SP,<Rm>{,<shift>}
```

15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0	1	1 1 0 1	S	1 1 0 1	(0)	imm3	Rd	imm2	type	Rm

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 && (shift_t != SRType_LSL || shift_n > 3) then UNPREDICTABLE;
if d == 15 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
SUB{S}<c> <Rd>,SP,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	0 0	0	0 0 1 0	S	1 1 0 1	Rd	imm5	type		0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SUB{S}<c><q>  {<Rd>,} SP, <Rm> {,<shift>}
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<Rd> — The destination register. If omitted, <Rd> is SP.

<Rm> — The register that is optionally shifted and used as the second operand.

<shift> — The shift to apply to the value read from <Rm>. If omitted, no shift is applied. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

In the Thumb instruction set, if <Rd> is SP or omitted, <shift> is only permitted to be omitted, LSL #1, LSL #2, or LSL #3.

The pre-UAL syntax SUB<c>S is equivalent to SUBS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(SP, NOT(shifted), '1');
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SUB{S}<c><q>  {<Rd>,} SP, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> — デスティネーションレジスタ。省略した場合、<Rd> として SP が使用されます。

<Rm> — オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ

<shift> — <Rm> から読み出された値に適用されるシフト。省略した場合、シフトは適用されません。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。

Thumb 命令セットでは、<Rd> として SP を使用した場合、または Rd を省略した場合、<shift> は省略、または LSL #1、LSL #2、LSL #3 のいずれかのみ許可されます。

UAL 以前の構文の SUB<c>S は、SUBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(SP, NOT(shifted), '1');
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.217 SUBS PC, LR and related instructions

These instructions are for system level use only. See *SUBS PC, LR and related instructions* on page B6-25.

A8.6.218 SVC (previously SWI)

Supervisor Call, previously called a Software Interrupt. For more information, see *Exceptions* on page B1-30, and in particular *Supervisor Call (SVC) exception* on page B1-52.

You can use this instruction as a call to an operating system to provide a service.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
SVC<c> #<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	imm8							

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly, and is ignored by hardware. SVC handlers in some
// systems interpret imm8 in software, for example to determine the required service.
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
SVC<c> #<imm24>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	1	imm24																							

```
imm32 = ZeroExtend(imm24, 32);
// imm32 is for assembly/disassembly, and is ignored by hardware. SVC handlers in some
// systems interpret imm24 in software, for example to determine the required service.
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.217 SUBS PC、LR、および関連する命令

これらの命令は、システムレベルでのみ使用できます。詳細については、*SUBS PC、LR、および関連する命令* P. B6-25 を参照して下さい。

A8.6.218 SVC（以前の SWI）

スーパバイザコールで、以前はソフトウェア割り込みと呼ばれていました。詳細については、*例外* P. B1-30、特に*スーパバイザコール (SVC) 例外* P. B1-52 を参照して下さい。

この命令は、サービスを提供するオペレーティングシステムへの呼び出しとして使用することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
SVC<c> #<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	imm8							

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly, and is ignored by hardware.SVC handlers in some
// systems interpret imm8 in software, for example to determine the required service.
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
SVC<c> #<imm24>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	1	imm24																							

```
imm32 = ZeroExtend(imm24, 32);
// imm32 is for assembly/disassembly, and is ignored by hardware.SVC handlers in some
// systems interpret imm24 in software, for example to determine the required service.
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SVC<c><q>  #<imm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<imm> Specifies an immediate constant, 8-bit in Thumb code, or 24-bit in ARM code.

The pre-UAL syntax SWI<c> is equivalent to SVC<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CallSupervisor();
```

Exceptions

Supervisor Call.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SVC<c><q>  #<imm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<imm> イミディエート定数を指定します。Thumb コードでは 8 ビット、ARM コードでは 24 ビットです。

UAL 以前の構文の SWI<c> は、SVC<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CallSupervisor();
```

例外

スーパバイザコール

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.219 SWP, SWPB

SWP (Swap) swaps a word between registers and memory. SWP loads a word from the memory address given by the value of register <Rn>. The value of register <Rt2> is then stored to the memory address given by the value of <Rn>, and the original loaded value is written to register <Rt>. If the same register is specified for <Rt> and <Rt2>, this instruction swaps the value of the register and the value at the memory address.

SWPB (Swap Byte) swaps a byte between registers and memory. SWPB loads a byte from the memory address given by the value of register <Rn>. The value of the least significant byte of register <Rt2> is stored to the memory address given by <Rn>, the original loaded value is zero-extended to a 32-bit word, and the word is written to register <Rt>. If the same register is specified for <Rt> and <Rt2>, this instruction swaps the value of the least significant byte of the register and the byte value at the memory address, and clears the most significant three bytes of the register.

For both instructions, the memory system ensures that no other memory access can occur to the memory location between the load access and the store access.

Note

- The SWP and SWPB instructions rely on the properties of the system beyond the processor to ensure that no stores from other observers can occur between the load access and the store access, and this might not be implemented for all regions of memory on some system implementations. In all cases, SWP and SWPB do ensure that no stores from the processor that executed the SWP or SWPB instruction can occur between the load access and the store access of the SWP or SWPB.
- The use of SWP is deprecated, and new code should use LDREX/STREX in preference to using SWP.
- The use of SWPB is deprecated, and new code should use LDREXB/STREXB in preference to using SWPB.

Encoding A1 ARMv4*, ARMv5T*, ARMv6* (deprecated), ARMv7 (deprecated)

```
SWP{B}<c> <Rt>,<Rt2>,[<Rn>]
```

31 30 29 28	27 26 25 24 23	22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1 0	B	0 0	Rn	Rt	(0) (0) (0) (0)	1 0 0 1	Rt2

```
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  size = if B == '1' then 1 else 4;
if t == 15 || t2 == 15 || n == 15 || n == t || n == t2 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.219 SWP、SWPB

SWP（スワップ）は、レジスタとメモリとの間で 1 ワードをスワップします。SWP は、レジスタ <Rn> の値によって示されるメモリアドレスから 1 ワードをロードします。次に、レジスタ <Rt2> の値が、<Rn> の値によって示されるメモリアドレスにストアされ、ロードされた元の値がレジスタ <Rt> に書き込まれます。<Rt> と <Rt2> で同じレジスタが指定されている場合、この命令は、そのレジスタの値と指定されたメモリアドレスの値とをスワップします。

SWPB（バイトスワップ）は、レジスタとメモリとの間で 1 バイトをスワップします。SWPB は、レジスタ <Rn> の値によって示されるメモリアドレスから 1 バイトをロードします。レジスタ <Rt2> の最下位バイトの値が、<Rn> によって示されるメモリアドレスにストアされ、ロードされた元の値が 32 ビットワードにゼロ拡張され、そのワードがレジスタ <Rt> に書き込まれます。<Rt> と <Rt2> で同じレジスタが指定されている場合、この命令は、そのレジスタの最下位バイトの値と、メモリアドレスのバイト値をスワップし、レジスタの上位 3 バイトをクリアします。

両方の命令について、メモリシステムによって、ロードアクセスとストアアクセスとの間でメモリ位置への他のメモリアクセスが発生しないことが保証されます。

注

- SWP 命令と SWPB 命令は、ロードアクセスとストアアクセスとの間で他のオブザーバからのストアが発生しないことを保証するため、プロセッサ以外のシステムのプロパティに依存します。一部のシステム実装では、これがすべてのメモリ領域で実装されているとは限りません。どのような場合でも、SWP 命令と SWPB 命令によって、SWP または SWPB を実行したプロセッサからのストアが、SWP または SWPB のロードアクセスとストアアクセスとの間で発生しないことは保証されます。
- SWP の使用は推奨されません。新しいコードでは、SWP よりも LDREX/STREX を使用するようにして下さい。
- SWPB の使用は推奨されません。新しいコードでは、SWPB よりも LDREXB/STREXB を使用するようにして下さい。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*（非推奨）、ARMv7（非推奨）

```
SWP{B}<c> <Rt>,<Rt2>,[<Rn>]
```

31 30 29 28	27 26 25 24 23	22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1 0	B	0 0	Rn	Rt	(0) (0) (0) (0)	1 0 0 1	Rt2

```
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  size = if B == '1' then 1 else 4;
if t == 15 || t2 == 15 || n == 15 || n == t || n == t2 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SWP{B}<c><q>  <Rt>, <Rt2>, [<Rn>]
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rt>`	The destination register.
`<Rt2>`	Contains the value that is stored to memory.
`<Rn>`	Contains the memory address to load from.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // The MemA[] accesses in the next two statements are locked together, that is, the memory
    // system must ensure that no other access to the same location can occur between them.
    data = MemA[R[n], size];
    MemA[R[n], size] = R[t2]<8*size-1:0>;
    if size == 1 then  // SWPB
        R[t] = ZeroExtend(data, 32);
    else               // SWP
        // Rotation in the following will always be by zero in ARMv7, due to alignment checks,
        // but can be nonzero in legacy configurations.
        R[t] = ROR(data, 8*UInt(address<1:0>));
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SWP{B}<c><q>  <Rt>, <Rt2>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rt>`	デスティネーションレジスタ
`<Rt2>`	メモリにストアされる値。
`<Rn>`	ロード元のメモリアドレス

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // The MemA[] accesses in the next two statements are locked together, that is, the memory
    // system must ensure that no other access to the same location can occur between them.
    data = MemA[R[n], size];
    MemA[R[n], size] = R[t2]<8*size-1:0>;
    if size == 1 then  // SWPB
        R[t] = ZeroExtend(data, 32);
    else               // SWP
        // Rotation in the following will always be by zero in ARMv7, due to alignment checks,
        // but can be nonzero in legacy configurations.
        R[t] = ROR(data, 8*UInt(address<1:0>));
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.220 SXTAB

Signed Extend and Add Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Encoding T1 ARMv6T2, ARMv7

```
SXTAB<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	0	Rn				1	1	1	1	Rd				1	(0)	rotate		Rm			

```
if Rn == '1111' then SEE SXTB;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SXTAB<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	0	Rn				Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
if Rn == '1111' then SEE SXTB;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.220 SXTAB

SXTAB（符号付きバイト拡張および加算）は、レジスタから 8 ビット値を抽出して、32 ビットに符号拡張し、結果を別のレジスタの値に加算し、最終結果をデスティネーションレジスタに書き込みます。8 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6T2、ARMv7

```
SXTAB<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	0	Rn				1	1	1	1	Rd				1	(0)	rotate		Rm			

```
if Rn == '1111' then SEE SXTB;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SXTAB<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	0	Rn				Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
if Rn == '1111' then SEE SXTB;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SXTAB<c><q>  {<Rd>,} <Rn>, <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rn>` The first operand register.

`<Rm>` The second operand register.

`<rotation>` This can be any one of:

omitted	encoded as rotate = '00'
`ROR #8`	encoded as rotate = '01'
`ROR #16`	encoded as rotate = '10'
`ROR #24`	encoded as rotate = '11'.

Note

An assembler can permit `ROR #0` to mean the same thing as omitting the rotation, possibly with restrictions on the permitted encodings, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<7:0>, 32);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SXTAB<c><q>  {<Rd>,} <Rn>, <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rn>` 最初のオペランドレジスタ

`<Rm>` 2 番目のオペランドレジスタ

`<rotation>` 次のいずれかを指定します。

省略	rotate = '00' としてエンコードされます。
`ROR #8`	rotate = '01' としてエンコードされます。
`ROR #16`	rotate = '10' としてエンコードされます。
`ROR #24`	rotate = '11' としてエンコードされます。

注

アセンブラは、`ROR #0` がローテーションの省略と同じ意味になることを許可できます（許可されるエンコードに対する制限を伴う場合があります）。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<7:0>, 32);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.221 SXTAB16

Signed Extend and Add Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

Encoding T1 ARMv6T2, ARMv7

```
SXTAB16<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	1	0	Rn				1	1	1	1	Rd				1	(0)	rotate		Rm			

```
if Rn == '1111' then SEE SXTB16;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SXTAB16<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	0	0	Rn				Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
if Rn == '1111' then SEE SXTB16;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.221 SXTAB16

SXTAB16（符号付きバイト拡張および加算 16）は、レジスタから 2 つの 8 ビット値を抽出して、それぞれ 16 ビットに符号拡張し、結果を別のレジスタからの 2 つの 16 ビット値に加算し、最終結果をデスティネーションレジスタに書き込みます。8 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6T2、ARMv7

```
SXTAB16<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	1	0	Rn				1	1	1	1	Rd				1	(0)	rotate		Rm			

```
if Rn == '1111' then SEE SXTB16;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SXTAB16<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	0	0	Rn				Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
if Rn == '1111' then SEE SXTB16;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SXTAB16<c><q>  {<Rd>,} <Rn>, <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rn>` The first operand register.

`<Rm>` The second operand register.

`<rotation>` This can be any one of:

omitted	encoded as rotate = '00'
`ROR #8`	encoded as rotate = '01'
`ROR #16`	encoded as rotate = '10'
`ROR #24`	encoded as rotate = '11'.

Note

An assembler can permit `ROR #0` to mean the same thing as omitting the rotation, possibly with restrictions on the permitted encodings, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = R[n]<15:0> + SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + SignExtend(rotated<23:16>, 16);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SXTAB16<c><q>  {<Rd>,} <Rn>, <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rn>` 最初のオペランドレジスタ

`<Rm>` 2 番目のオペランドレジスタ

`<rotation>` 次のいずれかを指定します。

省略	rotate = '00' としてエンコードされます。
`ROR #8`	rotate = '01' としてエンコードされます。
`ROR #16`	rotate = '10' としてエンコードされます。
`ROR #24`	rotate = '11' としてエンコードされます。

注

アセンブラは、`ROR #0` がローテーションの省略と同じ意味になることを許可できます（許可されるエンコードに対する制限を伴う場合があります）。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = R[n]<15:0> + SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + SignExtend(rotated<23:16>, 16);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.222 SXTAH

Signed Extend and Add Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Encoding T1 ARMv6T2, ARMv7

```
SXTAH<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	0	Rn				1	1	1	1	Rd				1	(0)	rotate		Rm			

```
if Rn == '1111' then SEE SXTH;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SXTAH<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	1	Rn				Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
if Rn == '1111' then SEE SXTH;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.222 SXTAH

SXTAH（符号付きハーフワード拡張および加算）は、レジスタから 16 ビット値を抽出して、32 ビットに符号拡張し、結果を別のレジスタからの値に加算し、最終結果をデスティネーションレジスタに書き込みます。16 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6T2、ARMv7

```
SXTAH<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	0	Rn				1	1	1	1	Rd				1	(0)	rotate		Rm			

```
if Rn == '1111' then SEE SXTH;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SXTAH<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	1	Rn				Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
if Rn == '1111' then SEE SXTH;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SXTAH<c><q>  {<Rd>,} <Rn>, <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rn>` The first operand register.

`<Rm>` The second operand register.

`<rotation>` This can be any one of:

omitted	encoded as rotate = '00'
`ROR #8`	encoded as rotate = '01'
`ROR #16`	encoded as rotate = '10'
`ROR #24`	encoded as rotate = '11'.

Note

An assembler can permit `ROR #0` to mean the same thing as omitting the rotation, possibly with restrictions on the permitted encodings, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<15:0>, 32);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SXTAH<c><q>  {<Rd>,} <Rn>, <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rn>` 最初のオペランドレジスタ

`<Rm>` 2 番目のオペランドレジスタ

`<rotation>` 次のいずれかを指定します。

省略	rotate = '00' としてエンコードされます。
`ROR #8`	rotate = '01' としてエンコードされます。
`ROR #16`	rotate = '10' としてエンコードされます。
`ROR #24`	rotate = '11' としてエンコードされます。

注

アセンブラは、`ROR #0` がローテーションの省略と同じ意味になることを許可できます（許可されるエンコードに対する制限を伴う場合があります）。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<15:0>, 32);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.223 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Encoding T1 ARMv6*, ARMv7

```
SXTB<c> <Rd>,<Rm>
```

15 14 13 12	11 10 9 8	7 6	5 4 3	2 1 0
1 0 1 1	0 0 1 0	0 1	Rm	Rd

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

Encoding T2 ARMv6T2, ARMv7

```
SXTB<c>.W <Rd>,<Rm>{,<rotation>}
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6	5 4	3 2 1 0
1 1 1 1 1	0 1 0	0	1 0 0	1 1 1 1	1 1 1 1	Rd	1	(0)	rotate	Rm

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SXTB<c> <Rd>,<Rm>{,<rotation>}
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10	9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 1 0 1 0	1 1 1 1	Rd	rotate	(0) (0)	0 1 1 1	Rm

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.223 SXTB

SXTB（符号付きバイト拡張）は、レジスタから 8 ビット値を抽出して、32 ビットに符号拡張し、結果をデスティネーションレジスタに書き込みます。8 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6*、ARMv7

```
SXTB<c> <Rd>,<Rm>
```

15 14 13 12	11 10 9 8	7 6	5 4 3	2 1 0
1 0 1 1	0 0 1 0	0 1	Rm	Rd

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

エンコード T2 ARMv6T2、ARMv7

```
SXTB<c>.W <Rd>,<Rm>{,<rotation>}
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6	5 4	3 2 1 0
1 1 1 1 1	0 1 0	0	1 0 0	1 1 1 1	1 1 1 1	Rd	1	(0)	rotate	Rm

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SXTB<c> <Rd>,<Rm>{,<rotation>}
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10	9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 1 0 1 0	1 1 1 1	Rd	rotate	(0) (0)	0 1 1 1	Rm

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SXTB<c><q>  {<Rd>,} <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rm>` The register that contains the operand.

`<rotation>` This can be any one of:

omitted any encoding, with rotate = '00' in encoding T2 or A1

`ROR #8` encoding T2 or A1, rotate = '01'

`ROR #16` encoding T2 or A1, rotate = '10'

`ROR #24` encoding T2 or A1, rotate = '11'.

Note

An assembler can permit `ROR #0` to mean the same thing as omitting the rotation, possibly with restrictions on the permitted encodings, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<7:0>, 32);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SXTB<c><q>  {<Rd>,} <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rm>` オペランドを含むレジスタ

`<rotation>` 次のいずれかを指定します。

省略 すべてのエンコード（エンコード T2 または A1 では rotate = '00'）

`ROR #8` エンコード T2 または A1、rotate = '01'

`ROR #16` エンコード T2 または A1、rotate = '10'

`ROR #24` エンコード T2 または A1、rotate = '11'

注

アセンブラは、`ROR #0` がローテーションの省略と同じ意味になることを許可できます（許可されるエンコードに対する制限を伴う場合があります）。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<7:0>, 32);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.224 SXTB16

Signed Extend Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, and writes the results to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

Encoding T1 ARMv6T2, ARMv7

```
SXTB16<c> <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SXTB16<c> <Rd>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	0	0	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.224 SXTB16

SXTB16（符号付きバイト拡張 16）は、レジスタから 2 つの 8 ビット値を抽出して、それぞれ 16 ビットに符号拡張し、結果をデスティネーションレジスタに書き込みます。8 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6T2、ARMv7

```
SXTB16<c> <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SXTB16<c> <Rd>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	0	0	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SXTB16<c><q>  {<Rd>,} <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rm>` The register that contains the operand.

`<rotation>` This can be any one of:

omitted	encoded as rotate = '00'
`ROR #8`	encoded as rotate = '01'
`ROR #16`	encoded as rotate = '10'
`ROR #24`	encoded as rotate = '11'.

Note

An assembler can permit `ROR #0` to mean the same thing as omitting the rotation, possibly with restrictions on the permitted encodings, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = SignExtend(rotated<23:16>, 16);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SXTB16<c><q>  {<Rd>,} <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rm>` オペランドを含むレジスタ

`<rotation>` 次のいずれかを指定します。

省略	rotate = '00' としてエンコードされます。
`ROR #8`	rotate = '01' としてエンコードされます。
`ROR #16`	rotate = '10' としてエンコードされます。
`ROR #24`	rotate = '11' としてエンコードされます。

注

アセンブラは、`ROR #0` がローテーションの省略と同じ意味になることを許可できます（許可されるエンコードに対する制限を伴う場合があります）。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = SignExtend(rotated<23:16>, 16);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.225 SXTH

Signed Extend Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Encoding T1 ARMv6*, ARMv7

```
SXTH<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	0	0	Rm				Rd	

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

Encoding T2 ARMv6T2, ARMv7

```
SXTH<c>.W <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SXTH<c> <Rd>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	1	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.225 SXTH

SXTH（符号付きハーフワード拡張）は、レジスタから 16 ビット値を抽出して、32 ビットに符号拡張し、結果をデスティネーションレジスタに書き込みます。16 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6*、ARMv7

```
SXTH<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	0	0	Rm				Rd	

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

エンコード T2 ARMv6T2、ARMv7

```
SXTH<c>.W <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SXTH<c> <Rd>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	0	1	1	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SXTH<c><q>  {<Rd>,} <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rm>` The register that contains the operand.

`<rotation>` This can be any one of:

omitted any encoding, with rotate = '00' in encoding T2 or A1

`ROR #8` encoding T2 or A1, rotate = ’01’

`ROR #16` encoding T2 or A1, rotate = ’10’

`ROR #24` encoding T2 or A1, rotate = ’11’.

Note

An assembler can permit `ROR #0` to mean the same thing as omitting the rotation, possibly with restrictions on the permitted encodings, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SXTH<c><q>  {<Rd>,} <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rm>` オペランドを含むレジスタ

`<rotation>` 次のいずれかを指定します。

省略 すべてのエンコード（エンコード T2 または A1 では rotate = ’00’）

`ROR #8` エンコード T2 または A1、rotate = ’01’

`ROR #16` エンコード T2 または A1、rotate = ’10’

`ROR #24` エンコード T2 または A1、rotate = ’11’

注

アセンブラは、`ROR #0` がローテーションの省略と同じ意味になることを許可できます（許可されるエンコードに対する制限を伴う場合があります）。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.226 TBB, TBH

Table Branch Byte causes a PC-relative forward branch using a table of single byte offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value of the byte returned from the table.

Table Branch Halfword causes a PC-relative forward branch using a table of single halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value of the halfword returned from the table.

Encoding T1 ARMv6T2, ARMv7

`TBB<c> [<Rn>,<Rm>]`	Outside or last in IT block
`TBH<c> [<Rn>,<Rm>,LSL #1]`	Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	H	Rm			

```
n = UInt(Rn);  m = UInt(Rm);  is_tbh = (H == '1');
if n == 13 || BadReg(m) then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.226 TBB、TBH

TBB（テーブル分岐バイト）は、単一バイトオフセットのテーブルを使用した、PC 相対の前方分岐を発生させます。ベースレジスタはテーブルへのポインタとして、2 番目のレジスタはテーブル内のインデクスとして使用されます。分岐の長さは、テーブルから返されたバイト値の 2 倍です。

TBH（テーブル分岐ハーフワード）は、単一ハーフワードオフセットのテーブルを使用した、PC 相対の前方分岐を発生させます。ベースレジスタはテーブルへのポインタとして、2 番目のレジスタはテーブル内のインデクスとして使用されます。分岐の長さは、テーブルから返されたハーフワード値の 2 倍です。

エンコード T1 ARMv6T2、ARMv7

`TBB<c> [<Rn>,<Rm>]`	IT ブロック外部または IT ブロック内部の最後
`TBH<c> [<Rn>,<Rm>,LSL #1]`	IT ブロック外部または IT ブロック内部の最後

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	H	Rm			

```
n = UInt(Rn);  m = UInt(Rm);  is_tbh = (H == '1');
if n == 13 || BadReg(m) then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
TBB<c><q>  [<Rn>, <Rm>]
TBH<c><q>  [<Rn>, <Rm>, LSL #1]
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The base register. This contains the address of the table of branch lengths. The PC can be used. If it is, the table immediately follows this instruction.

<Rm> The index register.

For TBB, this contains an integer pointing to a single byte in the table. The offset in the table is the value of the index.

For TBH, this contains an integer pointing to a halfword in the table. The offset in the table is twice the value of the index.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    if is_tbh then
        halfwords = UInt(MemU[R[n]+LSL(R[m],1), 2]);
    else
        halfwords = UInt(MemU[R[n]+R[m], 1]);
    BranchWritePC(PC + 2*halfwords);
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
TBB<c><q>  [<Rn>, <Rm>]
TBH<c><q>  [<Rn>, <Rm>, LSL #1]
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> ベースレジスタ。このレジスタには、分岐長のテーブルのアドレスが含まれます。このレジスタには PC を指定できます。その場合は、この命令の直後にテーブルが続きます。

<Rm> インデクスレジスタ

TBB の場合は、テーブル内の 1 バイトを指す整数が含まれます。テーブル内のオフセットはインデクスの値です。

TBH の場合は、テーブル内の 1 ハーフワードを指す整数が含まれます。テーブル内のオフセットはインデクスの値の 2 倍です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    if is_tbh then
        halfwords = UInt(MemU[R[n]+LSL(R[m],1), 2]);
    else
        halfwords = UInt(MemU[R[n]+R[m], 1]);
    BranchWritePC(PC + 2*halfwords);
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.227 TEQ (immediate)

Test Equivalence (immediate) performs a bitwise exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

Encoding T1 ARMv6T2, ARMv7

```
TEQ<c> <Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	1	0	0	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
TEQ<c> <Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	1	1	Rn				(0)	(0)	(0)	(0)	imm12											

```
n = UInt(Rn);
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.227 TEQ（イミディエート）

TEQ（等価テスト（イミディエート））は、レジスタの値とイミディエート値との間でビット単位の排他的論理和演算を実行します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv6T2、ARMv7

```
TEQ<c> <Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	1	0	0	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
TEQ<c> <Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	1	1	Rn				(0)	(0)	(0)	(0)	imm12											

```
n = UInt(Rn);
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
TEQ<c><q>  <Rn>, #<const>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The operand register.

<const> The immediate value to be tested against the value obtained from <Rn>. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
TEQ<c><q>  <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> オペランドレジスタ

<const> <Rn> から取得された値に対してテストされるイミディエート値。値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.228 TEQ (register)

Test Equivalence (register) performs a bitwise exclusive OR operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 ARMv6T2, ARMv7

```
TEQ<c> <Rn>,<Rm>{,<shift>}
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 1 0 0	1	Rn	(0)	imm3	1 1 1 1	imm2	type	Rm

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
TEQ<c> <Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
cond	0 0	0	1 0 0 1	1	Rn	(0) (0) (0) (0)	imm5	type	0	Rm

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.228 TEQ（レジスタ）

TEQ（等価テスト（レジスタ））は、レジスタの値と、オプションとしてシフトされたレジスタの値との間でビット単位の排他的論理和演算を実行します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv6T2、ARMv7

```
TEQ<c> <Rn>,<Rm>{,<shift>}
```

15 14 13 12 11	10 9	8 7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1	0 1	0 1 0 0	1	Rn	(0)	imm3	1 1 1 1	imm2	type	Rm

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
TEQ<c> <Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
cond	0 0	0	1 0 0 1	1	Rn	(0) (0) (0) (0)	imm5	type	0	Rm

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
TEQ<c><q>  <Rn>, <Rm> {,<shift>}
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The first operand register.

<Rm> The register that is optionally shifted and used as the second operand.

<shift> The shift to apply to the value read from <Rm>. If omitted, no shift is applied. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
TEQ<c><q>  <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

<Rn> 最初のオペランドレジスタ

<Rm> オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ

<shift> <Rm> から読み出された値に適用されるシフト。省略した場合、シフトは適用されません。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8- 10 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.229 TEQ (register-shifted register)

Test Equivalence (register-shifted register) performs a bitwise exclusive OR operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
TEQ<c> <Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 0 0 1	1	Rn	(0) (0) (0) (0)	Rs	0	type	1	Rm

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(type);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.229 TEQ（レジスタによりシフトしたレジスタ）

TEQ（等価テスト（レジスタによりシフトしたレジスタ））は、レジスタの値とレジスタによりシフトしたレジスタの値との間で、ビット単位の排他的論理和演算を実行します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
TEQ<c> <Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 0 0 1	1	Rn	(0) (0) (0) (0)	Rs	0	type	1	Rm

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(type);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
TEQ<c><q>  <Rn>, <Rm>, <type>  <Rs>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The first operand register.

<Rm> The register that is shifted and used as the second operand.

<type> The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR Arithmetic shift right, encoded as type = 0b10
- LSL Logical shift left, encoded as type = 0b00
- LSR Logical shift right, encoded as type = 0b01
- ROR Rotate right, encoded as type = 0b11.

<Rs> The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
TEQ<c><q>  <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> 最初のオペランドレジスタ

<Rm> シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR 算術右シフト。type = 0b10 としてエンコードされます。
- LSL 論理左シフト。type = 0b00 としてエンコードされます。
- LSR 論理右シフト。type = 0b01 としてエンコードされます。
- ROR 右ローテート。type = 0b11 としてエンコードされます。

<Rs> 最下位バイトにシフトのビット数を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.230 TST (immediate)

Test (immediate) performs a bitwise AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

Encoding T1 ARMv6T2, ARMv7

```
TST<c> <Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	0	0	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
TST<c> <Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	0	1	Rn				(0)	(0)	(0)	(0)	imm12											

```
n = UInt(Rn);
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.230 TST（イミディエート）

TST（テスト（イミディエート））は、レジスタの値とイミディエート値との間でビット単位の論理積演算を実行します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv6T2、ARMv7

```
TST<c> <Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	0	0	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
TST<c> <Rn>,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	0	1	Rn				(0)	(0)	(0)	(0)	imm12											

```
n = UInt(Rn);
(imm32, carry) = ARMExpandImm_C(imm12, APSR.C);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
TST<c><q>  <Rn>, #<const>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rn>` The operand register.

`<const>` The immediate value to be tested against the value obtained from `<Rn>`. See *Modified immediate constants in Thumb instructions* on page A6-17 or *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
TST<c><q>  <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rn>` オペランドレジスタ

`<const>` `<Rn>` から取得された値に対してテストされるイミディエート値。値の範囲については、*Thumb 命令での修飾イミディエート定数* P. A6-17 または *ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.231 TST (register)

Test (register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
TST<c> <Rn>,<Rm>
```

15 14 13 12 11 10	9 8 7 6	5 4 3	2 1 0
0 1 0 0 0 0	1 0 0 0	Rm	Rn

```
n = UInt(Rdn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv6T2, ARMv7

```
TST<c>.W  <Rn>,<Rm>{,<shift>}
```

15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1 0 1	0 0 0 0	1 Rn	(0) imm3	1 1 1 1	imm2	type	Rm

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
TST<c> <Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
cond	0 0	0	1 0 0 0	1	Rn	(0) (0) (0) (0)	imm5	type	0	Rm

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.231 TST（レジスタ）

TST（テスト（レジスタ））は、レジスタの値と、オプションとしてシフトされたレジスタの値との間でビット単位の論理積演算を実行します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

```
TST<c> <Rn>,<Rm>
```

15 14 13 12 11 10	9 8 7 6	5 4 3	2 1 0
0 1 0 0 0 0	1 0 0 0	Rm	Rn

```
n = UInt(Rdn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

```
TST<c>.W  <Rn>,<Rm>{,<shift>}
```

15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 0 1 0 1	0 0 0 0	1 Rn	(0) imm3	1 1 1 1	imm2	type	Rm

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
TST<c> <Rn>,<Rm>{,<shift>}
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
cond	0 0	0	1 0 0 0	1	Rn	(0) (0) (0) (0)	imm5	type	0	Rm

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
TST<c><q>  <Rn>, <Rm> {,<shift>}
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The first operand register.

<Rm> The register that is optionally shifted and used as the second operand.

<shift> The shift to apply to the value read from <Rm>. If present, encoding T1 is not permitted. If absent, no shift is applied and all encodings are permitted. *Shifts applied to a register* on page A8-10 describes the shifts and how they are encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
TST<c><q>  <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

<Rn> 最初のオペランドレジスタ

<Rm> オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタ

<shift> <Rm> から読み出された値に適用されるシフト。指定した場合、エンコード T1 は許可されません。省略した場合、シフトは適用されず、すべてのエンコードが許可されます。シフトとそのエンコード方法については、*レジスタに適用されるシフト* P. A8- 10 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.232 TST (register-shifted register)

Test (register-shifted register) performs a bitwise AND operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
TST<c> <Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 0 0 0	1	Rn	(0) (0) (0) (0)	Rs	0	type	1	Rm

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(type);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.232 TST（レジスタによりシフトしたレジスタ）

TST（テスト（レジスタによりシフトしたレジスタ））は、レジスタの値とレジスタによりシフトしたレジスタの値との間で、ビット単位の論理積演算を実行します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
TST<c> <Rn>,<Rm>,<type> <Rs>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	0 0	0	1 0 0 0	1	Rn	(0) (0) (0) (0)	Rs	0	type	1	Rm

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(type);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
TST<c><q>  <Rn>, <Rm>, <type>  <Rs>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The first operand register.

<Rm> The register that is shifted and used as the second operand.

<type> The type of shift to apply to the value read from <Rm>. It must be one of:

- ASR Arithmetic shift right, encoded as type = 0b10
- LSL Logical shift left, encoded as type = 0b00
- LSR Logical shift right, encoded as type = 0b01
- ROR Rotate right, encoded as type = 0b11.

<Rs> The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
TST<c><q>  <Rn>, <Rm>, <type>  <Rs>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> 最初のオペランドレジスタ

<Rm> シフトされ、2 番目のオペランドとして使用されるレジスタ

<type> <Rm> から読み出された値に適用されるシフトのタイプ。次のいずれかを使用できます。

- ASR 算術右シフト。type = 0b10 としてエンコードされます。
- LSL 論理左シフト。type = 0b00 としてエンコードされます。
- LSR 論理右シフト。type = 0b01 としてエンコードされます。
- ROR 右ローテート。type = 0b11 としてエンコードされます。

<Rs> 最下位バイトにシフトのビット数を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.233 UADD16

Unsigned Add 16 performs two 16-bit unsigned integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

Encoding T1 ARMv6T2, ARMv7

```
UADD16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	1 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UADD16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 0 1	Rn	Rd	(1) (1) (1) (1)	0 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.233 UADD16

UADD16（符号なし加算 16）は、2 つの符号なし 16 ビット整数加算を実行して、結果をデスティネーションレジスタに書き込みます。加算の結果に従って APSR.GE ビットをセットします。

エンコード T1 ARMv6T2、ARMv7

```
UADD16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	1 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UADD16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 0 1	Rn	Rd	(1) (1) (1) (1)	0 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UADD16<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<15:0>;
    R[d]<31:16> = sum2<15:0>;
    APSR.GE<1:0> = if sum1 >= 0x10000 then '11' else '00';
    APSR.GE<3:2> = if sum2 >= 0x10000 then '11' else '00';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UADD16<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<15:0>;
    R[d]<31:16> = sum2<15:0>;
    APSR.GE<1:0> = if sum1 >= 0x10000 then '11' else '00';
    APSR.GE<3:2> = if sum2 >= 0x10000 then '11' else '00';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.234 UADD8

Unsigned Add 8 performs four unsigned 8-bit integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

Encoding T1 ARMv6T2, ARMv7

```
UADD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	1 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UADD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 0 1	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.234 UADD8

UADD8（符号なし加算 8）は、4 つの符号なし 8 ビット整数加算を実行して、結果をデスティネーションレジスタに書き込みます。加算の結果に従って APSR.GE ビットをセットします。

エンコード T1 ARMv6T2、ARMv7

```
UADD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	1 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UADD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 0 1	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UADD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d]<7:0>   = sum1<7:0>;
    R[d]<15:8>  = sum2<7:0>;
    R[d]<23:16> = sum3<7:0>;
    R[d]<31:24> = sum4<7:0>;
    APSR.GE<0>  = if sum1 >= 0x100 then '1' else '0';
    APSR.GE<1>  = if sum2 >= 0x100 then '1' else '0';
    APSR.GE<2>  = if sum3 >= 0x100 then '1' else '0';
    APSR.GE<3>  = if sum4 >= 0x100 then '1' else '0';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UADD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d]<7:0>   = sum1<7:0>;
    R[d]<15:8>  = sum2<7:0>;
    R[d]<23:16> = sum3<7:0>;
    R[d]<31:24> = sum4<7:0>;
    APSR.GE<0>  = if sum1 >= 0x100 then '1' else '0';
    APSR.GE<1>  = if sum2 >= 0x100 then '1' else '0';
    APSR.GE<2>  = if sum3 >= 0x100 then '1' else '0';
    APSR.GE<3>  = if sum4 >= 0x100 then '1' else '0';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.235 UASX

Unsigned Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

Encoding T1 ARMv6T2, ARMv7

```
UASX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	1 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UASX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 0 1 0 1	Rn	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.235 UASX

UASX（交換付き符号なし加算および減算）は、2 番目のオペランドの 2 つのハーフワードを交換し、1 つの符号なし 16 ビット整数加算と 1 つの符号なし 16 ビット減算を実行して、結果をデスティネーションレジスタに書き込みます。結果に従って APSR.GE ビットをセットします。

エンコード T1 ARMv6T2、ARMv7

```
UASX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	1 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UASX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 0 1 0 1	Rn	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UASX<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

The pre-UAL syntax `UADDSUBX<c>` is equivalent to `UASX<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0>  = diff<15:0>;
    R[d]<31:16> = sum<15:0>;
    APSR.GE<1:0> = if diff >= 0 then '11' else '00';
    APSR.GE<3:2> = if sum  >= 0x10000 then '11' else '00';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UASX<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

UAL 以前の構文の `UADDSUBX<c>` は、`UASX<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0>  = diff<15:0>;
    R[d]<31:16> = sum<15:0>;
    APSR.GE<1:0> = if diff >= 0 then '11' else '00';
    APSR.GE<3:2> = if sum  >= 0x10000 then '11' else '00';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.236 UBFX

Unsigned Bit Field Extract extracts any number of adjacent bits at any position from a register, zero-extends them to 32 bits, and writes the result to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UBFX<c> <Rd>,<Rn>,#<lsb>,#<width>
```

15 14 13 12 11	10	9 8	7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5	4 3 2 1 0
1 1 1 1 0	(0)	1 1	1 1 0	0	Rn	0	imm3	Rd	imm2	(0)	widthm1

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv6T2, ARMv7

```
UBFX<c> <Rd>,<Rn>,#<lsb>,#<width>
```

31 30 29 28	27 26 25 24	23	22	21	20 19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 1	1	1	1	widthm1	Rd	lsb	1 0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(lsb);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.236 UBFX

UBFX（符号なしビットフィールドの抽出）は、1 つのレジスタの任意の位置にある、任意の数の隣接するビットを抽出して 32 ビットにゼロ拡張し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UBFX<c> <Rd>,<Rn>,#<lsb>,#<width>
```

15 14 13 12 11	10	9 8	7 6 5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5	4 3 2 1 0
1 1 1 1 0	(0)	1 1	1 1 0	0	Rn	0	imm3	Rd	imm2	(0)	widthm1

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv6T2、ARMv7

```
UBFX<c> <Rd>,<Rn>,#<lsb>,#<width>
```

31 30 29 28	27 26 25 24	23	22	21	20 19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 1	1	1	1	widthm1	Rd	lsb	1 0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(lsb);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UBFX<c><q>  <Rd>, <Rn>, #<lsb>, #<width>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<lsb> is the bit number of the least significant bit in the bitfield, in the range 0-31. This determines the required value of lsbit.

<width> is the width of the bitfield, in the range 1 to 32-<lsb>. The required value of widthminus1 is <width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = ZeroExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UBFX<c><q>  <Rd>, <Rn>, #<lsb>, #<width>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<lsb> ビットフィールド内の最下位ビットのビット番号で、範囲は 0 ～ 31 です。これによって、lsbit に必要な値が決定されます。

<width> ビットフィールドの幅で、範囲は 1 ～ 32 – <lsb> です。widthminus1 に必要な値は <width> - 1 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = ZeroExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.237 UDIV

Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition code flags are not affected.

Encoding T1 ARMv7-R

```
UDIV<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 1 0 1	1	Rn	(1) (1) (1) (1)	Rd	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.237 UDIV

UDIV（符号なし除算）は、32 ビットの符号なし整数レジスタの値を、32 ビットの符号なし整数レジスタの値で除算して、結果をデスティネーションレジスタに書き込みます。条件コードフラグは影響を受けません。

エンコード T1 ARMv7-R

```
UDIV<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 1 0 1	1	Rn	(1) (1) (1) (1)	Rd	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UDIV<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The register that contains the dividend.
`<Rm>`	The register that contains the divisor.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if UInt(R[m]) == 0 then
        if IntegerZeroDivideTrappingEnabled() then
            GenerateIntegerZeroDivide();
        else
            result = 0;
    else
        result = RoundTowardsZero(UInt(R[n]) / UInt(R[m]));
    R[d] = result<31:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UDIV<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	被除数を含むレジスタ
`<Rm>`	除数を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if UInt(R[m]) == 0 then
        if IntegerZeroDivideTrappingEnabled() then
            GenerateIntegerZeroDivide();
        else
            result = 0;
    else
        result = RoundTowardsZero(UInt(R[n]) / UInt(R[m]));
    R[d] = result<31:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.238 UHADD16

Unsigned Halving Add 16 performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UHADD16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1	0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	1 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UHADD16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 1	Rn	Rd	(1) (1) (1) (1)	0 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.238 UHADD16

UHADD16（符号なし加算半減 16）は、2 つの符号なし 16 ビット整数加算を実行し、結果を半減し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UHADD16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1	0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	1 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UHADD16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 1	Rn	Rd	(1) (1) (1) (1)	0 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UHADD16<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<16:1>;
    R[d]<31:16> = sum2<16:1>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UHADD16<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<16:1>;
    R[d]<31:16> = sum2<16:1>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.239 UHADD8

Unsigned Halving Add 8 performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UHADD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	1 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UHADD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 1	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.239 UHADD8

UHADD8（符号なし加算半減 8）は、4 つの符号なし 8 ビット整数加算を実行し、結果を半減し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UHADD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	1 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UHADD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 1	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UHADD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d]<7:0>   = sum1<8:1>;
    R[d]<15:8>  = sum2<8:1>;
    R[d]<23:16> = sum3<8:1>;
    R[d]<31:24> = sum4<8:1>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UHADD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d]<7:0>   = sum1<8:1>;
    R[d]<15:8>  = sum2<8:1>;
    R[d]<23:16> = sum3<8:1>;
    R[d]<31:24> = sum4<8:1>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.240 UHASX

Unsigned Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, halves the results, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UHASX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	1 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UHASX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 1	Rn	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.240 UHASX

UHASX（交換付き符号なし加算および減算半減）は、2 番目のオペランドの 2 つのハーフワードを交換し、1 つの符号なし 16 ビット整数加算と 1 つの符号なし 16 ビット減算を実行し、結果を半減し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UHASX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	1 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UHASX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 1	Rn	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UHASX<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<Rm> The second operand register.

The pre-UAL syntax UHADDSUBX<c> is equivalent to UHASX<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0>  = diff<16:1>;
    R[d]<31:16> = sum<16:1>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UHASX<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<Rm> 2 番目のオペランドレジスタ

UAL 以前の構文の UHADDSUBX<c> は、UHASX<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0>  = diff<16:1>;
    R[d]<31:16> = sum<16:1>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.241 UHSAX

Unsigned Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, halves the results, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UHSAX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 1 0	Rn	1 1 1 1	Rd	0	1 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UHSAX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 1	Rn	Rd	(1) (1) (1) (1)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.241 UHSAX

UHSAX（交換付き符号なし減算および加算半減）は、2 番目のオペランドの 2 つのハーフワードを交換し、1 つの符号なし 16 ビット整数減算と 1 つの符号なし 16 ビット加算を実行し、結果を半減し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UHSAX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 1 0	Rn	1 1 1 1	Rd	0	1 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UHSAX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 1	Rn	Rd	(1) (1) (1) (1)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UHSAX<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<Rm> The second operand register.

The pre-UAL syntax UHSUBADDX<c> is equivalent to UHSAX<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0>  = sum<16:1>;
    R[d]<31:16> = diff<16:1>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UHSAX<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<Rm> 2 番目のオペランドレジスタ

UAL 以前の構文の UHSUBADDX<c> は、UHSAX<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0>  = sum<16:1>;
    R[d]<31:16> = diff<16:1>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.242 UHSUB16

Unsigned Halving Subtract 16 performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UHSUB16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	1 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UHSUB16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 1	Rn	Rd	(1) (1) (1) (1)	0	1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.242 UHSUB16

UHSUB16（符号なし減算半減 16）は、2 つの符号なし 16 ビット整数減算を実行し、結果を半減し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UHSUB16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	1 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UHSUB16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 1	Rn	Rd	(1) (1) (1) (1)	0	1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UHSUB16<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<16:1>;
    R[d]<31:16> = diff2<16:1>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UHSUB16<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<16:1>;
    R[d]<31:16> = diff2<16:1>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.243 UHSUB8

Unsigned Halving Subtract 8 performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UHSUB8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 0	Rn	1 1 1 1	Rd	0	1 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UHSUB8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 1	Rn	Rd	(1) (1) (1) (1)	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.243 UHSUB8

UHSUB8（符号なし減算半減 8）は、4 つの符号なし 8 ビット整数減算を実行し、結果を半減し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UHSUB8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 0	Rn	1 1 1 1	Rd	0	1 1 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UHSUB8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 1	Rn	Rd	(1) (1) (1) (1)	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UHSUB8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]<7:0>   = diff1<8:1>;
    R[d]<15:8>  = diff2<8:1>;
    R[d]<23:16> = diff3<8:1>;
    R[d]<31:24> = diff4<8:1>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UHSUB8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]<7:0>   = diff1<8:1>;
    R[d]<15:8>  = diff2<8:1>;
    R[d]<23:16> = diff3<8:1>;
    R[d]<31:24> = diff4<8:1>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.244 UMAAL

Unsigned Multiply Accumulate Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, adds two unsigned 32-bit values, and writes the 64-bit result to two registers.

Encoding T1 ARMv6T2, ARMv7

```
UMAAL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 1	1 1 0	Rn	RdLo	RdHi	0 1 1 0	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UMAAL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 0 1 0 0	RdHi	RdLo	Rm	1 0 0 1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.244 UMAAL

UMAAL（符号なしロング積和和演算）は、2 つの符号なし 32 ビット値を乗算して 64 ビット値を生成し、2 つの符号なし 32 ビット値を加算し、64 ビットの結果を 2 つのレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UMAAL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 1	1 1 0	Rn	RdLo	RdHi	0 1 1 0	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UMAAL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 0 1 0 0	RdHi	RdLo	Rm	1 0 0 1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UMAAL<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<RdLo>` Supplies one of the 32 bit values to be added, and is the destination register for the lower 32 bits of the result.

`<RdHi>` Supplies the other of the 32 bit values to be added, and is the destination register for the upper 32 bits of the result.

`<Rn>` The register that contains the first multiply operand.

`<Rm>` The register that contains the second multiply operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]) + UInt(R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UMAAL<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<RdLo>` 加算される一方の 32 ビット値で、同時に結果の下位 32 ビットのデスティネーションレジスタとなります。

`<RdHi>` 加算される他方の 32 ビット値で、同時に結果の上位 32 ビットのデスティネーションレジスタとなります。

`<Rn>` 最初の乗算オペランドを含むレジスタ

`<Rm>` 2 番目の乗算オペランドを含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]) + UInt(R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.245 UMLAL

Unsigned Multiply Accumulate Long multiplies two unsigned32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In ARM code, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many processor implementations.

Encoding T1 ARMv6T2, ARMv7

```
UMLAL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 1	1 1 0	Rn	RdLo	RdHi	0 0 0 0	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
UMLAL{S}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 1 0 1	S	RdHi	RdLo	Rm	1 0 0 1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
if ArchVersion() < 6 && (dHi == n || dLo == n) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.245 UMLAL

UMLAL（符号なしロング積和演算）は、2 つの符号なし 32 ビット値を乗算して 64 ビット値を生成し、別の 64 ビット値と加算します。

ARM コードでは、オプションとして、結果に基づいて条件フラグを更新することができます。このオプションを使用すると、多くのプロセッサ実装で性能が低下します。

エンコード T1 ARMv6T2、ARMv7

```
UMLAL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 1	1 1 0	Rn	RdLo	RdHi	0 0 0 0	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
UMLAL{S}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 1 0 1	S	RdHi	RdLo	Rm	1 0 0 1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
if ArchVersion() < 6 && (dHi == n || dLo == n) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UMLAL{S}<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

where:

S — If S is present, the instruction updates the flags. Otherwise, the flags are not updated.
S can be specified only for the ARM instruction set.

<c><q> — See *Standard assembler syntax fields* on page A8-7.

<RdLo> — Supplies the lower 32 bits of the accumulate value, and is the destination register for the lower 32 bits of the result.

<RdHi> — Supplies the upper 32 bits of the accumulate value, and is the destination register for the upper 32 bits of the result.

<Rn> — The first operand register.

<Rm> — The second operand register.

The pre-UAL syntax UMLAL<c>S is equivalent to UMLALS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        APSR.N = result<63>;
        APSR.Z = IsZeroBit(result<63:0>);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
            APSR.V = bit UNKNOWN;
        // else APSR.C, APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UMLAL{S}<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S — S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。
S は、ARM 命令セットでのみ指定できます。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<RdLo> — 加算される値の下位 32 ビットで、同時に結果の下位 32 ビットのデスティネーションレジスタとなります。

<RdHi> — 加算される値の上位 32 ビットで、同時に結果の上位 32 ビットのデスティネーションレジスタとなります。

<Rn> — 最初のオペランドレジスタ

<Rm> — 2 番目のオペランドレジスタ

UAL 以前の構文の UMLAL<c>S は、UMLALS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        APSR.N = result<63>;
        APSR.Z = IsZeroBit(result<63:0>);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
            APSR.V = bit UNKNOWN;
        // else APSR.C, APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.246 UMULL

Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result.

In ARM code, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many processor implementations.

Encoding T1 ARMv6T2, ARMv7

```
UMULL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 1	0 1 0	Rn	RdLo	RdHi	0 0 0 0	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
UMULL{S}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 1 0 0	S	RdHi	RdLo	Rm	1 0 0 1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
if ArchVersion() < 6 && (dHi == n || dLo == n) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.246 UMULL

UMULL（符号なしロング乗算）は、2 つの符号なし 32 ビット値を乗算して、64 ビットの結果を生成します。

ARM コードでは、オプションとして、結果に基づいて条件フラグを更新することができます。このオプションを使用すると、多くのプロセッサ実装で性能が低下します。

エンコード T1 ARMv6T2、ARMv7

```
UMULL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 1	0 1 0	Rn	RdLo	RdHi	0 0 0 0	Rm

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
UMULL{S}<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 0 1 0 0	S	RdHi	RdLo	Rm	1 0 0 1	Rn

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
if ArchVersion() < 6 && (dHi == n || dLo == n) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UMULL{S}<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

where:

S	If S is present, the instruction updates the flags. Otherwise, the flags are not updated. S can be specified only for the ARM instruction set.
<c><q>	See *Standard assembler syntax fields* on page A8-7.
<RdLo>	Stores the lower 32 bits of the result.
<RdHi>	Stores the upper 32 bits of the result.
<Rn>	The first operand register.
<Rm>	The second operand register.

The pre-UAL syntax UMULL<c>S is equivalent to UMULLS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        APSR.N = result<63>;
        APSR.Z = IsZeroBit(result<63:0>);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
            APSR.V = bit UNKNOWN;
        // else APSR.C, APSR.V unchanged
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UMULL{S}<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S	S を指定した場合、命令によってフラグが更新されます。その他の場合、フラグは更新されません。 S は、ARM 命令セットでのみ指定できます。
<c><q>	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
<RdLo>	結果の下位 32 ビットが格納されます。
<RdHi>	結果の上位 32 ビットが格納されます。
<Rn>	最初のオペランドレジスタ
<Rm>	2 番目のオペランドレジスタ

UAL 以前の構文の UMULL<c>S は、UMULLS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        APSR.N = result<63>;
        APSR.Z = IsZeroBit(result<63:0>);
        if ArchVersion() == 4 then
            APSR.C = bit UNKNOWN;
            APSR.V = bit UNKNOWN;
        // else APSR.C, APSR.V unchanged
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.247 UQADD16

Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range $0 \le x \le 2^{16} - 1$, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UQADD16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1	0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UQADD16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 0	Rn	Rd	(1) (1) (1) (1)	0 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.247 UQADD16

UQADD16（符号なし飽和加算 16）は、2 つの符号なし 16 ビット整数加算を実行し、結果を符号なし 16 ビット整数の範囲 ($0 \leqq x \leqq 2^{16} - 1$) に飽和させ、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UQADD16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1	0 1 0	1	0 0 1	Rn	1 1 1 1	Rd	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UQADD16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 0	Rn	Rd	(1) (1) (1) (1)	0 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UQADD16<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0>  = UnsignedSat(sum1, 16);
    R[d]<31:16> = UnsignedSat(sum2, 16);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
UQADD16<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0>  = UnsignedSat(sum1, 16);
    R[d]<31:16> = UnsignedSat(sum2, 16);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.248 UQADD8

Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range $0 \le x \le 2^8 - 1$, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UQADD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UQADD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 0	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.248 UQADD8

UQADD8（符号なし飽和加算 8）は、4 つの符号なし 8 ビット整数加算を実行し、結果を符号なし 8 ビット整数の範囲 ($0 \leqq x \leqq 2^8 - 1$) に飽和させ、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UQADD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	0 0 0	Rn	1 1 1 1	Rd	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UQADD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 0	Rn	Rd	(1) (1) (1) (1)	1 0 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UQADD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d]<7:0>   = UnsignedSat(sum1, 8);
    R[d]<15:8>  = UnsignedSat(sum2, 8);
    R[d]<23:16> = UnsignedSat(sum3, 8);
    R[d]<31:24> = UnsignedSat(sum4, 8);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UQADD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d]<7:0>   = UnsignedSat(sum1, 8);
    R[d]<15:8>  = UnsignedSat(sum2, 8);
    R[d]<23:16> = UnsignedSat(sum3, 8);
    R[d]<31:24> = UnsignedSat(sum4, 8);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.249 UQASX

Unsigned Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturates the results to the 16-bit unsigned integer range $0 \le x \le 2^{16} - 1$, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UQASX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1	0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UQASX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 0 1 1 0	Rn	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.249 UQASX

UQASX（交換付き符号なし飽和加算および減算）は、2 番目のオペランドの 2 つのハーフワードを交換し、1 つの符号なし 16 ビット整数加算と 1 つの符号なし 16 ビット減算を実行し、結果を符号なし 16 ビット整数の範囲 ($0 \leqq x \leqq 2^{16} - 1$) に飽和させ、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UQASX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1	0 1 0	1	0 1 0	Rn	1 1 1 1	Rd	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UQASX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 0 1 1 0	Rn	Rd	(1) (1) (1) (1)	0 0 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UQASX<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<Rm> The second operand register.

The pre-UAL syntax UQADDSUBX<c> is equivalent to UQASX<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0>  = UnsignedSat(diff, 16);
    R[d]<31:16> = UnsignedSat(sum, 16);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UQASX<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<Rm> 2 番目のオペランドレジスタ

UAL 以前の構文の UQADDSUBX<c> は、UQASX<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0>  = UnsignedSat(diff, 16);
    R[d]<31:16> = UnsignedSat(sum, 16);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.250 UQSAX

Unsigned Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturates the results to the 16-bit unsigned integer range $0 \le x \le 2^{16} - 1$, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UQSAX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 1 0	Rn	1 1 1 1	Rd	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UQSAX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 0 1 1 0	Rn	Rd	(1) (1) (1) (1)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.250 UQSAX

UQSAX（交換付き符号なし飽和減算および加算）は、2 番目のオペランドの 2 つのハーフワードを交換し、1 つの符号なし 16 ビット整数減算と 1 つの符号なし 16 ビット加算を実行し、結果を符号なし 16 ビット整数の範囲 ($0 \leqq x \leqq 2^{16} - 1$) に飽和させ、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UQSAX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 1 0	Rn	1 1 1 1	Rd	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UQSAX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0 0 1 1 0	Rn	Rd	(1) (1) (1) (1)	0 1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UQSAX<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rn>` The first operand register.

`<Rm>` The second operand register.

The pre-UAL syntax `UQSUBADDX<c>` is equivalent to `UQSAX<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0>  = UnsignedSat(sum, 16);
    R[d]<31:16> = UnsignedSat(diff, 16);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UQSAX<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rn>` 最初のオペランドレジスタ

`<Rm>` 2 番目のオペランドレジスタ

UAL 以前の構文の `UQSUBADDX<c>` は、`UQSAX<c>` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0>  = UnsignedSat(sum, 16);
    R[d]<31:16> = UnsignedSat(diff, 16);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.251 UQSUB16

Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range $0 \le x \le 2^{16} - 1$, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UQSUB16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1	0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UQSUB16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 0	Rn	Rd	(1) (1) (1) (1)	0 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.251 UQSUB16

UQSUB16（符号なし飽和減算 16）は、2 つの符号なし 16 ビット整数減算を実行し、結果を符号なし 16 ビット整数の範囲 (0 ≦ x ≦ 2^{16} - 1) に飽和させ、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UQSUB16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1	0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UQSUB16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 0	Rn	Rd	(1) (1) (1) (1)	0 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UQSUB16<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0>  = UnsignedSat(diff1, 16);
    R[d]<31:16> = UnsignedSat(diff2, 16);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UQSUB16<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0>  = UnsignedSat(diff1, 16);
    R[d]<31:16> = UnsignedSat(diff2, 16);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.252 UQSUB8

Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range $0 \le x \le 2^8 - 1$, and writes the results to the destination register.

Encoding T1 ARMv6T2, ARMv7

```
UQSUB8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 0	Rn	1 1 1 1	Rd	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UQSUB8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 0	Rn	Rd	(1) (1) (1) (1)	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.252 UQSUB8

UQSUB8（符号なし飽和減算 8）は、4 つの符号なし 8 ビット整数減算を実行し、結果を符号なし 8 ビット整数の範囲 ($0 \leqq x \leqq 2^8 - 1$) に飽和させ、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

```
UQSUB8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 0	Rn	1 1 1 1	Rd	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UQSUB8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 1 0	Rn	Rd	(1) (1) (1) (1)	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UQSUB8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]<7:0>   = UnsignedSat(diff1, 8);
    R[d]<15:8>  = UnsignedSat(diff2, 8);
    R[d]<23:16> = UnsignedSat(diff3, 8);
    R[d]<31:24> = UnsignedSat(diff4, 8);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UQSUB8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]<7:0>   = UnsignedSat(diff1, 8);
    R[d]<15:8>  = UnsignedSat(diff2, 8);
    R[d]<23:16> = UnsignedSat(diff3, 8);
    R[d]<31:24> = UnsignedSat(diff4, 8);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.253 USAD8

Unsigned Sum of Absolute Differences performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

Encoding T1 ARMv6T2, ARMv7

```
USAD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 0	1 1 1	Rn	1 1 1 1	Rd	0 0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
USAD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 1	1 0 0 0	Rd	1 1 1 1	Rm	0 0 0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.253 USAD8

USAD8（符号なし絶対差の和）は、4 つの符号なし 8 ビット減算を実行し、それぞれの差の絶対値を加算します。

エンコード T1 ARMv6T2、ARMv7

```
USAD8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 0	1 1 1	Rn	1 1 1 1	Rd	0 0 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
USAD8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 1	1 0 0 0	Rd	1 1 1 1	Rm	0 0 0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
USAD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    result = absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
USAD8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    result = absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.254 USADA8

Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

Encoding T1 ARMv6T2, ARMv7

```
USADA8<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 0	1 1 1	Rn	Ra	Rd	0 0 0 0	Rm

```
if Ra == '1111' then SEE USAD8;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a == UInt(Ra);
if BadReg(d) || BadReg(n) || BadReg(m) || BadReg(a) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
USADA8<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 1	1 0 0 0	Rd	Ra	Rm	0 0 0 1	Rn

```
if Ra == '1111' then SEE USAD8;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a == UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.254 USADA8

USADA8（符号なし絶対差の和および累積）は、4 つの符号なし 8 ビット減算を実行し、それぞれの差の絶対値を 32 ビット累積オペランドに加算します。

エンコード T1 ARMv6T2、ARMv7

```
USADA8<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15 14 13 12 11	10 9 8 7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1	0 1 1 0	1 1 1	Rn	Ra	Rd	0 0 0 0	Rm

```
if Ra == '1111' then SEE USAD8;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a == UInt(Ra);
if BadReg(d) || BadReg(n) || BadReg(m) || BadReg(a) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
USADA8<c> <Rd>,<Rn>,<Rm>,<Ra>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 1	1 0 0 0	Rd	Ra	Rm	0 0 0 1	Rn

```
if Ra == '1111' then SEE USAD8;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a == UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
USADA8<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.
`<Ra>`	The register that contains the accumulation value.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    result = UInt(R[a]) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
USADA8<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ
`<Ra>`	累積値を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    result = UInt(R[a]) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.255 USAT

Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range.

The Q flag is set if the operation saturates.

Encoding T1 ARMv6T2, ARMv7

```
USAT<c> <Rd>,#<imm5>,<Rn>{,<shift>}
```

15 14 13 12	11	10	9 8	7 6	5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
1 1 1 1	0	(0)	1 1	1 0	sh	0	Rn	0	imm3	Rd	imm2	(0)		sat_imm

```
if sh == '1' && (imm3:imm2) == '00000' then SEE USAT16;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
USAT<c> <Rd>,#<imm5>,<Rn>{,<shift>}
```

31 30 29 28	27 26 25 24	23 22 21	20 19 18 17 16	15 14 13 12	11 10 9 8	7	6	5 4	3 2 1 0
cond	0 1 1 0	1 1 1	sat_imm	Rd	imm5		sh	0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.255 USAT

USAT（符号なし飽和）は、オプションとしてシフトされた符号付きの値を、選択された符号なし範囲に飽和させます。

演算が飽和した場合、Q フラグがセットされます。

エンコード T1 ARMv6T2、ARMv7

```
USAT<c> <Rd>,#<imm5>,<Rn>{,<shift>}
```

15 14 13 12	11	10	9 8	7 6	5	4	3 2 1 0	15	14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
1 1 1 1	0	(0)	1 1	1 0	sh	0	Rn	0	imm3	Rd	imm2	(0)		sat_imm

```
if sh == '1' && (imm3:imm2) == '00000' then SEE USAT16;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
USAT<c> <Rd>,#<imm5>,<Rn>{,<shift>}
```

31 30 29 28	27 26 25 24	23 22 21	20 19 18 17 16	15 14 13 12	11 10 9 8	7	6	5 4	3 2 1 0
cond	0 1 1 0	1 1 1	sat_imm	Rd	imm5		sh	0 1	Rn

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
USAT<c><q>  <Rd>, #<imm>, <Rn> {,<shift>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<imm>` The bit position for saturation, in the range 0 to 31.

`<Rn>` The register that contains the value to be saturated.

`<shift>` The optional shift, encoded in the sh bit and five bits in imm3:imm2 for encoding T1 and imm5 for encoding A1. It must be one of:

omitted No shift. Encoded as sh = 0, five bits = `0b00000`

`LSL #<n>` Left shift by `<n>` bits, with `<n>` in the range 1-31.
Encoded as sh = 0, five bits = `<n>`.

`ASR #<n>` Arithmetic right shift by `<n>` bits, with `<n>` in the range 1-31.
Encoded as sh = 1, five bits = `<n>`.

`ASR #32` Arithmetic right shift by 32 bits, permitted only for encoding A1.
Encoded as sh = 1, imm5 = `0b00000`.

Note

An assembler can permit `ASR #0` or `LSL #0` to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, APSR.C);  // APSR.C ignored
    (result, sat) = UnsignedSatQ(SInt(operand), saturate_to);
    R[d] = ZeroExtend(result, 32);
    if sat then
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
USAT<c><q>  <Rd>, #<imm>, <Rn> {,<shift>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<imm>` 飽和のビット位置（0 ～ 31）。

`<Rn>` 飽和される値を含むレジスタ

`<shift>` オプションのシフト。sh ビットと、エンコード T1 では imm3:imm2、エンコード A1 では imm5 の 5 ビットにエンコードされます。次のいずれかを使用できます。

省略 シフトなし。sh = 0、5 つのビット = `0b00000` としてエンコードされます。

`LSL #<n>` `<n>` ビットだけ左シフトが適用されます。`<n>` の範囲は 1 ～ 31 です。
sh = 0、5 つのビット = `<n>` としてエンコードされます。

`ASR #<n>` `<n>` ビットだけ算術右シフトが適用されます。`<n>` の範囲は 1 ～ 31 です。
sh = 1、5 つのビット = `<n>` としてエンコードされます。

`ASR #32` 32 ビットだけ算術右シフトが適用されます。エンコード A1 でのみ許可されます。
sh = 1、imm5 = `0b00000` としてエンコードされます。

注

アセンブラは、`ASR #0` または `LSL #0` がシフトの省略と同じ意味になることを許可できます。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, APSR.C);  // APSR.C ignored
    (result, sat) = UnsignedSatQ(SInt(operand), saturate_to);
    R[d] = ZeroExtend(result, 32);
    if sat then
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.256 USAT16

Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range.

The Q flag is set if the operation saturates.

Encoding T1 ARMv6T2, ARMv7

```
USAT16<c> <Rd>,#<imm4>,<Rn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	1	0	1	0	Rn				0	0	0	0	Rd				0	0	(0)	(0)	sat_imm			

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
USAT16<c> <Rd>,#<imm4>,<Rn>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	1	1	0	sat_imm				Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rn			

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if d == 15 || n == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.256 USAT16

USAT16（符号なし飽和 16）は、2 つの符号付き 16 ビット値を、選択された符号なし範囲に飽和させます。

演算が飽和した場合、Q フラグがセットされます。

エンコード T1 ARMv6T2、ARMv7

```
USAT16<c> <Rd>,#<imm4>,<Rn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	1	0	1	0	Rn				0	0	0	0	Rd				0	0	(0)	(0)	sat_imm			

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
USAT16<c> <Rd>,#<imm4>,<Rn>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	1	1	0	sat_imm				Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rn			

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if d == 15 || n == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
USAT16<c><q>  <Rd>, #<imm>, <Rn>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<imm>`	The bit position for saturation, in the range 0 to 15.
`<Rn>`	The register that contains the values to be saturated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = UnsignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = UnsignedSatQ(SInt(R[n]<31:16>), saturate_to);
    R[d]<15:0> = ZeroExtend(result1, 16);
    R[d]<31:16> = ZeroExtend(result2, 16);
    if sat1 || sat2 then
        APSR.Q = '1';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
USAT16<c><q>  <Rd>, #<imm>, <Rn>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<imm>`	飽和のビット位置（0 ～ 15）
`<Rn>`	飽和される値を含むレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = UnsignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = UnsignedSatQ(SInt(R[n]<31:16>), saturate_to);
    R[d]<15:0> = ZeroExtend(result1, 16);
    R[d]<31:16> = ZeroExtend(result2, 16);
    if sat1 || sat2 then
        APSR.Q = '1';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.257 USAX

Unsigned Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

Encoding T1 ARMv6T2, ARMv7

```
USAX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 1 0	Rn	1 1 1 1	Rd	0	1 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
USAX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 1 0 1	Rn	Rd	(1) (1) (1) (1)	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.257 USAX

USAX（交換付き符号なし減算および加算）は、2 番目のオペランドの 2 つのハーフワードを交換し、1 つの符号なし 16 ビット整数減算と 1 つの符号なし 16 ビット加算を実行して、結果をデスティネーションレジスタに書き込みます。結果に従って APSR.GE ビットをセットします。

エンコード T1 ARMv6T2、ARMv7

```
USAX<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 1 0	Rn	1 1 1 1	Rd	0	1 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
USAX<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 1 0 1	Rn	Rd	(1) (1) (1) (1)	0	1 0 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
USAX<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rd> The destination register.

<Rn> The first operand register.

<Rm> The second operand register.

The pre-UAL syntax USUBADDX<c> is equivalent to USAX<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0>  = sum<15:0>;
    R[d]<31:16> = diff<15:0>;
    APSR.GE<1:0> = if sum  >= 0x10000 then '11' else '00';
    APSR.GE<3:2> = if diff >= 0 then '11' else '00';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
USAX<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rd> デスティネーションレジスタ

<Rn> 最初のオペランドレジスタ

<Rm> 2 番目のオペランドレジスタ

UAL 以前の構文の USUBADDX<c> は、USAX<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0>  = sum<15:0>;
    R[d]<31:16> = diff<15:0>;
    APSR.GE<1:0> = if sum  >= 0x10000 then '11' else '00';
    APSR.GE<3:2> = if diff >= 0 then '11' else '00';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.258 USUB16

Unsigned Subtract 16 performs two 16-bit unsigned integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

Encoding T1 ARMv6T2, ARMv7

```
USUB16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	1 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
USUB16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 1 0 1	Rn	Rd	(1) (1) (1) (1)	0	1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.258 USUB16

USUB16（符号なし減算 16）は、2 つの符号なし 16 ビット整数減算を実行して、結果をデスティネーションレジスタに書き込みます。減算の結果に従って APSR.GE ビットをセットします。

エンコード T1 ARMv6T2、ARMv7

```
USUB16<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 1	Rn	1 1 1 1	Rd	0	1 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
USUB16<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0	0 1 0 1	Rn	Rd	(1) (1) (1) (1)	0	1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
USUB16<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<15:0>;
    R[d]<31:16> = diff2<15:0>;
    APSR.GE<1:0> = if diff1 >= 0 then '11' else '00';
    APSR.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
USUB16<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<15:0>;
    R[d]<31:16> = diff2<15:0>;
    APSR.GE<1:0> = if diff1 >= 0 then '11' else '00';
    APSR.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.259 USUB8

Unsigned Subtract 8 performs four 8-bit unsigned integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

Encoding T1 ARMv6T2, ARMv7

```
USUB8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 0	Rn	1 1 1 1	Rd	0	1 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
USUB8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 0 1	Rn	Rd	(1) (1) (1) (1)	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.259 USUB8

USUB8（符号なし減算 8）は、4 つの符号なし 8 ビット整数減算を実行して、結果をデスティネーションレジスタに書き込みます。減算の結果に従って APSR.GE ビットをセットします。

エンコード T1 ARMv6T2、ARMv7

```
USUB8<c> <Rd>,<Rn>,<Rm>
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
1 1 1 1 1	0 1 0	1	1 0 0	Rn	1 1 1 1	Rd	0	1 0 0	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
USUB8<c> <Rd>,<Rn>,<Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 1 1 0	0 1 0 1	Rn	Rd	(1) (1) (1) (1)	1 1 1 1	Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
USUB8<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Rd>`	The destination register.
`<Rn>`	The first operand register.
`<Rm>`	The second operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]<7:0>   = diff1<7:0>;
    R[d]<15:8>  = diff2<7:0>;
    R[d]<23:16> = diff3<7:0>;
    R[d]<31:24> = diff4<7:0>;
    APSR.GE<0>  = if diff1 >= 0 then '1' else '0';
    APSR.GE<1>  = if diff2 >= 0 then '1' else '0';
    APSR.GE<2>  = if diff3 >= 0 then '1' else '0';
    APSR.GE<3>  = if diff4 >= 0 then '1' else '0';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
USUB8<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Rd>`	デスティネーションレジスタ
`<Rn>`	最初のオペランドレジスタ
`<Rm>`	2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]<7:0>   = diff1<7:0>;
    R[d]<15:8>  = diff2<7:0>;
    R[d]<23:16> = diff3<7:0>;
    R[d]<31:24> = diff4<7:0>;
    APSR.GE<0>  = if diff1 >= 0 then '1' else '0';
    APSR.GE<1>  = if diff2 >= 0 then '1' else '0';
    APSR.GE<2>  = if diff3 >= 0 then '1' else '0';
    APSR.GE<3>  = if diff4 >= 0 then '1' else '0';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.260 UXTAB

Unsigned Extend and Add Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Encoding T1 ARMv6T2, ARMv7

```
UXTAB<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	1	Rn				1	1	1	1	Rd				1	(0)	rotate		Rm			

```
if Rn == '1111' then SEE UXTB;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UXTAB<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	1	1	0	Rn				Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
if Rn == '1111' then SEE UXTB;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.260 UXTAB

UXTAB（符号なしバイト拡張および加算）は、レジスタから 8 ビット値を抽出して、32 ビットにゼロ拡張し、結果を別のレジスタの値に加算し、最終結果をデスティネーションレジスタに書き込みます。8 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6T2、ARMv7

```
UXTAB<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	1	Rn				1	1	1	1	Rd				1	(0)	rotate		Rm			

```
if Rn == '1111' then SEE UXTB;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UXTAB<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	1	1	0	Rn				Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
if Rn == '1111' then SEE UXTB;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UXTAB<c><q>  {<Rd>,} <Rn>, <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rn>` The first operand register.

`<Rm>` The second operand register.

`<rotation>` This can be any one of:

omitted	encoded as rotate = '00'
`ROR #8`	encoded as rotate = '01'
`ROR #16`	encoded as rotate = '10'
`ROR #24`	encoded as rotate = '11'.

Note

An assembler can permit `ROR #0` to mean the same thing as omitting the rotation, possibly with restrictions on the permitted encodings, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<7:0>, 32);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UXTAB<c><q>  {<Rd>,} <Rn>, <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rn>` 最初のオペランドレジスタ

`<Rm>` 2 番目のオペランドレジスタ

`<rotation>` 次のいずれかを指定します。

省略	rotate = '00' としてエンコードされます。
`ROR #8`	rotate = '01' としてエンコードされます。
`ROR #16`	rotate = '10' としてエンコードされます。
`ROR #24`	rotate = '11' としてエンコードされます。

注

アセンブラは、`ROR #0` がローテーションの省略と同じ意味になることを許可できます（許可されるエンコードに対する制限を伴う場合があります）。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<7:0>, 32);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.261 UXTAB16

Unsigned Extend and Add Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

Encoding T1 ARMv6T2, ARMv7

```
UXTAB16<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6	5 4	3 2 1 0
1 1 1 1 1	0 1 0	0	0 1 1	Rn	1 1 1 1	Rd	1	(0)	rotate	Rm

```
if Rn == '1111' then SEE UXTB16;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UXTAB16<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10	9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0 1 1 0 0	Rn	Rd	rotate	(0) (0)	0	1 1 1	Rm

```
if Rn == '1111' then SEE UXTB16;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.261 UXTAB16

UXTAB16（符号なしバイト拡張および加算 16）は、レジスタから 2 つの 8 ビット値を抽出して、それぞれ 16 ビットにゼロ拡張し、結果を別のレジスタからの 2 つの 16 ビット値に加算し、最終結果をデスティネーションレジスタに書き込みます。8 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6T2、ARMv7

```
UXTAB16<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

15 14 13 12 11	10 9 8	7	6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7	6	5 4	3 2 1 0
1 1 1 1 1	0 1 0	0	0 1 1	Rn	1 1 1 1	Rd	1	(0)	rotate	Rm

```
if Rn == '1111' then SEE UXTB16;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UXTAB16<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12	11 10	9 8	7	6 5 4	3 2 1 0
cond	0 1 1 0 1 1 0 0	Rn	Rd	rotate	(0) (0)	0	1 1 1	Rm

```
if Rn == '1111' then SEE UXTB16;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UXTAB16<c><q>  {<Rd>,} <Rn>, <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rn>` The first operand register.

`<Rm>` The second operand register.

`<rotation>` This can be any one of:

omitted encoded as rotate = '00'

`ROR #8` encoded as rotate = '01'

`ROR #16` encoded as rotate = '10'

`ROR #24` encoded as rotate = '11'.

Note

An assembler can permit `ROR #0` to mean the same thing as omitting the rotation, possibly with restrictions on the permitted encodings, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = R[n]<15:0> + ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + ZeroExtend(rotated<23:16>, 16);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UXTAB16<c><q>  {<Rd>,} <Rn>, <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rn>` 最初のオペランドレジスタ

`<Rm>` 2 番目のオペランドレジスタ

`<rotation>` 次のいずれかを指定します。

省略 rotate = '00' としてエンコードされます。

`ROR #8` rotate = '01' としてエンコードされます。

`ROR #16` rotate = '10' としてエンコードされます。

`ROR #24` rotate = '11' としてエンコードされます。

注

アセンブラは、`ROR #0` がローテーションの省略と同じ意味になることを許可できます（許可されるエンコードに対する制限を伴う場合があります）。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = R[n]<15:0> + ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + ZeroExtend(rotated<23:16>, 16);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.262 UXTAH

Unsigned Extend and Add Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Encoding T1 ARMv6T2, ARMv7

```
UXTAH<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	1	Rn				1	1	1	1	Rd				1	(0)	rotate		Rm			

```
if Rn == '1111' then SEE UXTH;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UXTAH<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	1	1	1	Rn				Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
if Rn == '1111' then SEE UXTH;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.262 UXTAH

UXTAH（符号なしハーフワード拡張および加算）は、レジスタから 16 ビット値を抽出して、32 ビットにゼロ拡張し、結果を別のレジスタからの値に加算し、最終結果をデスティネーションレジスタに書き込みます。16 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6T2、ARMv7

```
UXTAH<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	1	Rn				1	1	1	1	Rd				1	(0)	rotate		Rm			

```
if Rn == '1111' then SEE UXTH;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UXTAH<c> <Rd>,<Rn>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	1	1	1	Rn				Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
if Rn == '1111' then SEE UXTH;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UXTAH<c><q>  {<Rd>,} <Rn>, <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rn>` The first operand register.

`<Rm>` The second operand register.

`<rotation>` This can be any one of:

omitted	encoded as rotate = '00'
`ROR #8`	encoded as rotate = '01'
`ROR #16`	encoded as rotate = '10'
`ROR #24`	encoded as rotate = '11'.

Note

An assembler can permit `ROR #0` to mean the same thing as omitting the rotation, possibly with restrictions on the permitted encodings, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<15:0>, 32);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UXTAH<c><q>  {<Rd>,} <Rn>, <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rn>` 最初のオペランドレジスタ

`<Rm>` 2 番目のオペランドレジスタ

`<rotation>` 次のいずれかを指定します。

省略	rotate = '00' としてエンコードされます。
`ROR #8`	rotate = '01' としてエンコードされます。
`ROR #16`	rotate = '10' としてエンコードされます。
`ROR #24`	rotate = '11' としてエンコードされます。

注

アセンブラは、`ROR #0` がローテーションの省略と同じ意味になることを許可できます（許可されるエンコードに対する制限を伴う場合があります）。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<15:0>, 32);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.263 UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Encoding T1 ARMv6*, ARMv7

```
UXTB<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	1	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

Encoding T2 ARMv6T2, ARMv7

```
UXTB<c>.W <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UXTB<c> <Rd>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	1	1	0	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.263 UXTB

UXTB（符号なしバイト拡張）は、レジスタから 8 ビット値を抽出して、32 ビットにゼロ拡張し、結果をデスティネーションレジスタに書き込みます。8 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6*、ARMv7

```
UXTB<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	1	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

エンコード T2 ARMv6T2、ARMv7

```
UXTB<c>.W <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UXTB<c> <Rd>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	1	1	0	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UXTB<c><q>  {<Rd>,} <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rm>` The second operand register.

`<rotation>` This can be any one of:

omitted any encoding, with rotate = '00' in encoding T2 or A1

`ROR #8` encoding T2 or A1, rotate = '01'

`ROR #16` encoding T2 or A1, rotate = '10'

`ROR #24` encoding T2 or A1, rotate = '11'.

Note

An assembler can permit `ROR #0` to mean the same thing as omitting the rotation, possibly with restrictions on the permitted encodings, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UXTB<c><q>  {<Rd>,} <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rm>` 2 番目のオペランドレジスタ

`<rotation>` 次のいずれかを指定します。

省略 すべてのエンコード（エンコード T2 または A1 では rotate = '00'）

`ROR #8` エンコード T2 または A1、rotate = '01'

`ROR #16` エンコード T2 または A1、rotate = '10'

`ROR #24` エンコード T2 または A1、rotate = '11'

注

アセンブラは、`ROR #0` がローテーションの省略と同じ意味になることを許可できます（許可されるエンコードに対する制限を伴う場合があります）。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.264 UXTB16

Unsigned Extend Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, and writes the results to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

Encoding T1 ARMv6T2, ARMv7

```
UXTB16<c> <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UXTB16<c> <Rd>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	1	0	0	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.264 UXTB16

UXTB16（符号なしバイト拡張 16）は、レジスタから 2 つの 8 ビット値を抽出して、それぞれ 16 ビットにゼロ拡張し、結果をデスティネーションレジスタに書き込みます。8 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6T2、ARMv7

```
UXTB16<c> <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UXTB16<c> <Rd>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	1	0	0	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UXTB16<c><q>  {<Rd>,} <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rm>` The second operand register.

`<rotation>` This can be any one of:

omitted	encoded as rotate = '00'
`ROR #8`	encoded as rotate = '01'
`ROR #16`	encoded as rotate = '10'
`ROR #24`	encoded as rotate = '11'.

Note

An assembler can permit `ROR #0` to mean the same thing as omitting the rotation, possibly with restrictions on the permitted encodings, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = ZeroExtend(rotated<23:16>, 16);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UXTB16<c><q>  {<Rd>,} <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rm>` 2 番目のオペランドレジスタ

`<rotation>` 次のいずれかを指定します。

省略	rotate = '00' としてエンコードされます。
`ROR #8`	rotate = '01' としてエンコードされます。
`ROR #16`	rotate = '10' としてエンコードされます。
`ROR #24`	rotate = '11' としてエンコードされます。

注

アセンブラは、`ROR #0` がローテーションの省略と同じ意味になることを許可できます（許可されるエンコードに対する制限を伴う場合があります）。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = ZeroExtend(rotated<23:16>, 16);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.265 UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Encoding T1 ARMv6*, ARMv7

```
UXTH<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	1	0	Rm				Rd	

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

Encoding T2 ARMv6T2, ARMv7

```
UXTH<c>.W <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
UXTH<c> <Rd>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	1	1	1	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.265 UXTH

UXTH（符号なしハーフワード拡張）は、レジスタから 16 ビット値を抽出して、32 ビットにゼロ拡張し、結果をデスティネーションレジスタに書き込みます。16 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6*、ARMv7

```
UXTH<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	1	0	Rm				Rd	

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

エンコード T2 ARMv6T2、ARMv7

```
UXTH<c>.W <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
UXTH<c> <Rd>,<Rm>{,<rotation>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	1	1	0	1	1	1	1	1	1	1	1	Rd				rotate		(0)	(0)	0	1	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
UXTH<c><q>  {<Rd>,} <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<Rm>` The second operand register.

`<rotation>` This can be any one of:

omitted any encoding, with rotate = '00' in encoding T2 or A1

`ROR #8` encoding T2 or A1, rotate = '01'

`ROR #16` encoding T2 or A1, rotate = '10'

`ROR #24` encoding T2 or A1, rotate = '11'.

Note

An assembler can permit `ROR #0` to mean the same thing as omitting the rotation, possibly with restrictions on the permitted encodings, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
UXTH<c><q>  {<Rd>,} <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<Rm>` 2 番目のオペランドレジスタ

`<rotation>` 次のいずれかを指定します。

省略 すべてのエンコード（エンコード T2 または A1 では rotate = '00'）

`ROR #8` エンコード T2 または A1、rotate = '01'

`ROR #16` エンコード T2 または A1、rotate = '10'

`ROR #24` エンコード T2 または A1、rotate = '11'

注

アセンブラは、`ROR #0` がローテーションの省略と同じ意味になることを許可できます（許可されるエンコードに対する制限を伴う場合があります）。しかし、これは標準的な UAL ではなく、逆アセンブリには使用できません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.266 VABA, VABAL

Vector Absolute Difference and Accumulate {Long} subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand and result elements are either all integers of the same length, or optionally the results can be double the length of the operands.

Encoding T1 / A1 Advanced SIMD

```
VABA<c>.<dt>  <Qd>, <Qn>, <Qm>
VABA<c>.<dt>  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	1	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	1	1	N	Q	M	1	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 Advanced SIMD

```
VABAL<c>.<dt>  <Qd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				0	1	0	1	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				0	1	0	1	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.266 VABA、VABAL

VABA、VABAL（ベクタ絶対差および加算ロング）は、一方のベクタのエレメントを他方のベクタの対応するエレメントから減算し、結果の絶対値をデスティネーションベクタのエレメントに加算します。

オペランドと結果のエレメントは、すべて整数で同じ長さです。ただしオプションとして、結果をオペランドの長さの 2 倍とすることができます。

エンコード T1/A1 アドバンスト SIMD

```
VABA<c>.<dt>  <Qd>, <Qn>, <Qm>
VABA<c>.<dt>  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	1	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	1	1	N	Q	M	1	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 アドバンスト SIMD

```
VABAL<c>.<dt>  <Qd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				0	1	0	1	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				0	1	0	1	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VABA<c><q>.<dt> <Qd>, <Qn>, <Qm>`	Encoding T1 / A1, Q = 1
`VABA<c><q>.<dt> <Dd>, <Dn>, <Dm>`	Encoding T1 / A1, Q = 0
`VABAL<c><q>.<dt> <Qd>, <Dn>, <Dm>`	Encoding T2 / A2

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VABA` or `VABAL` instruction must be unconditional.

`<dt>` The data type for the elements of the operands. It must be one of:

- `S8` encoded as size = 0b00, U = 0
- `S16` encoded as size = 0b01, U = 0
- `S32` encoded as size = 0b10, U = 0
- `U8` encoded as size = 0b00, U = 1
- `U16` encoded as size = 0b01, U = 1
- `U32` encoded as size = 0b10, U = 1.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

`<Qd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a long operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];
            op2 = Elem[D[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Q[d>>1],e,2*esize] + absdiff;
            else
                Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + absdiff;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VABA<c><q>.<dt> <Qd>, <Qn>, <Qm>`	エンコード T1/A1、Q = 1
`VABA<c><q>.<dt> <Dd>, <Dn>, <Dm>`	エンコード T1/A1、Q = 0
`VABAL<c><q>.<dt> <Qd>, <Dn>, <Dm>`	エンコード T2/A2

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VABA` および `VABAL` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S8` size = 0b00、U = 0 としてエンコードされます。
- `S16` size = 0b01、U = 0 としてエンコードされます。
- `S32` size = 0b10、U = 0 としてエンコードされます。
- `U8` size = 0b00、U = 1 としてエンコードされます。
- `U16` size = 0b01、U = 1 としてエンコードされます。
- `U32` size = 0b10、U = 1 としてエンコードされます。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Qd>, <Dn>, <Dm>` ロング演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];
            op2 = Elem[D[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Q[d>>1],e,2*esize] + absdiff;
            else
                Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + absdiff;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.267 VABD, VABDL (integer)

Vector Absolute Difference {Long} (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are either all integers of the same length, or optionally the results can be double the length of the operands.

Encoding T1 / A1 Advanced SIMD

```
VABD<c>.<dt>  <Qd>, <Qn>, <Qm>
VABD<c>.<dt>  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	1	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	1	1	N	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 Advanced SIMD

```
VABDL<c>.<dt>  <Qd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				0	1	1	1	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				0	1	1	1	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.267 VABD、VABDL（整数）

VABD、VABDL（ベクタ絶対差ロング（整数））は、一方のベクタのエレメントを他方のベクタの対応するエレメントから減算し、結果の絶対値をデスティネーションベクタのエレメントに配置します。

オペランドと結果のエレメントは、すべて整数で同じ長さです。ただしオプションとして、結果をオペランドの長さの 2 倍とすることができます。

エンコード T1/A1 アドバンスト SIMD

```
VABD<c>.<dt>  <Qd>, <Qn>, <Qm>
VABD<c>.<dt>  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	1	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	1	1	N	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 アドバンスト SIMD

```
VABDL<c>.<dt>  <Qd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				0	1	1	1	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				0	1	1	1	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VABD<c><q>.<dt> <Qd>, <Qn>, <Qm>`	Encoding T1 / A1, Q = 1
`VABD<c><q>.<dt> <Dd>, <Dn>, <Dm>`	Encoding T1 / A1, Q = 0
`VABDL<c><q>.<dt> <Qd>, <Dn>, <Dm>`	Encoding T2 / A2

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VABD` or `VABDL` instruction must be unconditional.

`<dt>` The data type for the elements of the operands. It must be one of:

- `S8` encoded as size = 0b00, U = 0
- `S16` encoded as size = 0b01, U = 0
- `S32` encoded as size = 0b10, U = 0
- `U8` encoded as size = 0b00, U = 1
- `U16` encoded as size = 0b01, U = 1
- `U32` encoded as size = 0b10, U = 1.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

`<Qd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a long operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];
            op2 = Elem[D[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
            else
                Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VABD<c><q>.<dt> <Qd>, <Qn>, <Qm>`	エンコード T1/A1、Q = 1
`VABD<c><q>.<dt> <Dd>, <Dn>, <Dm>`	エンコード T1/A1、Q = 0
`VABDL<c><q>.<dt> <Qd>, <Dn>, <Dm>`	エンコード T2/A2

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VABD` および `VABDL` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S8` size = 0b00、U = 0 としてエンコードされます。
- `S16` size = 0b01、U = 0 としてエンコードされます。
- `S32` size = 0b10、U = 0 としてエンコードされます。
- `U8` size = 0b00、U = 1 としてエンコードされます。
- `U16` size = 0b01、U = 1 としてエンコードされます。
- `U32` size = 0b10、U = 1 としてエンコードされます。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Qd>, <Dn>, <Dm>` ロング演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];
            op2 = Elem[D[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
            else
                Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.268 VABD (floating-point)

Vector Absolute Difference (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are all single-precision floating-point numbers.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
VABD<c>.F32  <Qd>, <Qn>, <Qm>
VABD<c>.F32  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.268 VABD（浮動小数点）

VABD（ベクタ絶対差（浮動小数点））は、一方のベクタのエレメントを他方のベクタの対応するエレメントから減算し、結果の絶対値をデスティネーションベクタのエレメントに配置します。

オペランドと結果のエレメントはすべて単精度浮動小数点数値です。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VABD<c>.F32  <Qd>, <Qn>, <Qm>
VABD<c>.F32  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VABD<c><q>.F32 <Qd>, <Qn>, <Qm>`	Encoded as Q = 1, sz = 0
`VABD<c><q>.F32 <Dd>, <Dn>, <Dm>`	Encoded as Q = 0, sz = 0

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM `VABD` instruction must be unconditional.
`<Qd>, <Qn>, <Qm>`	The destination vector and the operand vectors, for a quadword operation.
`<Dd>, <Dn>, <Dm>`	The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            Elem[D[d+r],e,esize] = FPAbs(FPSub(op1,op2,FALSE));
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, Overflow, Underflow, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VABD<c><q>.F32 <Qd>, <Qn>, <Qm>`	Q = 1、sz = 0 としてエンコードされる
`VABD<c><q>.F32 <Dd>, <Dn>, <Dm>`	Q = 0、sz = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VABD` は常に無条件命令です。
`<Qd>, <Qn>, <Qm>`	クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Dd>, <Dn>, <Dm>`	ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            Elem[D[d+r],e,esize] = FPAbs(FPSub(op1,op2,FALSE));
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、オーバフロー、アンダーフロー、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.269 VABS

Vector Absolute takes the absolute value of each element in a vector, and places the results in a second vector. The floating-point version only clears the sign bit.

Encoding T1 / A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

```
VABS<c>.<dt> <Qd>, <Qm>
VABS<c>.<dt> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	1	1	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	1	1	0	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VABS<c>.F64 <Dd>, <Dm>
VABS<c>.F32 <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	1	sz	1	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	1	sz	1	1	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP vectors Encoding T2 / A2 can operate on VFP vectors under control of the FPSCR.LEN and FPSCR.STRIDE bits. For details see Appendix F *VFP Vector Operation Support*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.269 VABS

VABS（ベクタ絶対値）は、ベクタの各エレメントの絶対値を取得し、結果を 2 番目のベクタに配置します。浮動小数点バージョンは、符号ビットのクリアのみを行います。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは F = 1 は未定義）

```
VABS<c>.<dt> <Qd>, <Qm>
VABS<c>.<dt> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	1	1	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	1	1	0	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VABS<c>.F64 <Dd>, <Dm>
VABS<c>.F32 <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	1	sz	1	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	1	sz	1	1	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP ベクタ エンコード T2/A2 は、FPSCR.LEN ビットと FPSCR.STRIDE ビットの制御の下で VFP ベクタに作用することができます。詳細については、付録 F 「*VFP ベクタ演算のサポート*」を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VABS<c><q>.<dt> <Qd>, <Qm>`	`<dt>` != `F64`
`VABS<c><q>.<dt> <Dd>, <Dm>`	
`VABS<c><q>.F32 <Sd>, <Sm>`	VFP only. Encoding T2/A2, sz = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VABS` instruction must be unconditional.

`<dt>` The data type for the elements of the vectors. It must be one of:

- `S8` encoding T1 / A1, size = 0b00, F = 0
- `S16` encoding T1 / A1, size = 0b01, F = 0
- `S32` encoding T1 / A1, size = 0b10, F = 0
- `F32` encoding T1 / A1, size = 0b10, F = 1
- `F64` encoding T2 / A2, sz = 1.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

`<Sd>, <Sm>` The destination vector and the operand vector, for a singleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPAbs(Elem[D[m+r],e,esize]);
                else
                    result = Abs(SInt(Elem[D[m+r],e,esize]));
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
    else             // VFP instruction
        if dp_operation then
            D[d] = FPAbs(D[m]);
        else
            S[d] = FPAbs(S[m]);
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VABS<c><q>.<dt> <Qd>, <Qm>`	`<dt>` != `F64`
`VABS<c><q>.<dt> <Dd>, <Dm>`	
`VABS<c><q>.F32 <Sd>, <Sm>`	VFP のみ。エンコード T2/A2、sz = 0

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VABS` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S8` エンコード T1/A1、size = 0b00、F = 0
- `S16` エンコード T1/A1、size = 0b01、F = 0
- `S32` エンコード T1/A1、size = 0b10、F = 0
- `F32` エンコード T1/A1、size = 0b10、F = 1
- `F64` エンコード T2/A2、sz = 1

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Sd>, <Sm>` シングルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPAbs(Elem[D[m+r],e,esize]);
                else
                    result = Abs(SInt(Elem[D[m+r],e,esize]));
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
    else             // VFP instruction
        if dp_operation then
            D[d] = FPAbs(D[m]);
        else
            S[d] = FPAbs(S[m]);
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.270 VACGE, VACGT, VACLE,VACLT

VACGE (Vector Absolute Compare Greater Than or Equal) and VACGT (Vector Absolute Compare Greater Than) take the absolute value of each element in a vector, and compare it with the absolute value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

VACLE (Vector Absolute Compare Less Than or Equal) is a pseudo-instruction, equivalent to a VACGE instruction with the operands reversed. Disassembly produces the VACGE instruction.

VACLT (Vector Absolute Compare Less Than) is a pseudo-instruction, equivalent to a VACGT instruction with the operands reversed. Disassembly produces the VACGT instruction.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

The operand vector elements must be 32-bit floating-point numbers.

The result vector elements are 32-bit bitfields.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
V<op><c>.F32 <Qd>, <Qn>, <Qm>
V<op><c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	op	sz	Vn				Vd				1	1	1	0	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	op	sz	Vn				Vd				1	1	1	0	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
or_equal = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.270 VACGE、VACGT、VACLE、VACLT

VACGE（ベクタ絶対値比較以上）およびVACGT（ベクタ絶対値比較より大きい）は、ベクタの各エレメントの絶対値を取得して、2 番目のベクタの対応するエレメントの絶対値と比較します。条件が真の場合、デスティネーションベクタの対応するエレメントがすべて 1 にセットされます。それ以外の場合、すべて 0 にセットされます。

VACLE（ベクタ絶対値比較以下）は擬似命令で、オペランドが反転された VACGE 命令と等価です。VACGE 命令は、逆アセンブリによって生成されます。

VACLT（ベクタ絶対値比較未満）は擬似命令で、オペランドが反転された VACGT 命令と等価です。VACGT 命令は、逆アセンブリによって生成されます。

オペランドと結果は、クワッドワードまたはダブルワードのベクタです。オペランドと結果は、すべて同じサイズの必要があります。

オペランドベクタのエレメントは、32 ビットの浮動小数点数値の必要があります。

結果ベクタのエレメントは、32 ビットのビットフィールドです。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
V<op><c>.F32 <Qd>, <Qn>, <Qm>
V<op><c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	op	sz	Vn				Vd				1	1	1	0	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	op	sz	Vn				Vd				1	1	1	0	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
or_equal = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`V<op><c><q>.F32 {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`V<op><c><q>.F32 {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<op>` The operation. `<op>` must be one of:

- `ACGE` Absolute Compare Greater than or Equal, encoded as op = 0
- `ACGT` Absolute Compare Greater Than, encoded as op = 1.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VACGE`, `VACGT`, `VACLE`, or `VACLT` instruction must be unconditional.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = FPAbs(Elem[D[n+r],e,esize]);  op2 = FPAbs(Elem[D[m+r],e,esize]);
            if or_equal then
                test_passed = FPCompareGE(op1, op2, FALSE);
            else
                test_passed = FPCompareGT(op1, op2, FALSE);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal and Invalid Operation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`V<op><c><q>.F32 {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`V<op><c><q>.F32 {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<op>` 演算。`<op>` は次のいずれかです。

- `ACGE` 絶対比較以上（op = 0 としてエンコード）
- `ACGT` 絶対比較より大きい（op = 1 としてエンコード）

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VACGE`、`VACGT`、`VACLE`、および `VACLT` は常に無条件命令です。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = FPAbs(Elem[D[n+r],e,esize]);  op2 = FPAbs(Elem[D[m+r],e,esize]);
            if or_equal then
                test_passed = FPCompareGE(op1, op2, FALSE);
            else
                test_passed = FPCompareGT(op1, op2, FALSE);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.271 VADD (integer)

Vector Add adds corresponding elements in two vectors, and places the results in the destination vector.

Encoding T1 / A1 Advanced SIMD

```
VADD<c>.<dt> <Qd>, <Qn>, <Qm>
VADD<c>.<dt> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.271 VADD（整数）

VADD（ベクタ加算（整数））は、2 つのベクタの対応するエレメントを加算し、結果をデスティネーションベクタに配置します。

エンコード T1/A1 アドバンスト SIMD

```
VADD<c>.<dt> <Qd>, <Qn>, <Qm>
VADD<c>.<dt> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VADD<c><q>.<dt>  {<Qd>,} <Qn>, <Qm>
VADD<c><q>.<dt>  {<Dd>,} <Dn>, <Dm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VADD` instruction must be unconditional.

`<dt>` The data type for the elements of the vectors. It must be one of:

`I8`	size = 0b00
`I16`	size = 0b01
`I32`	size = 0b10
`I64`	size = 0b11.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] + Elem[D[m+r],e,esize];
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VADD<c><q>.<dt>  {<Qd>,} <Qn>, <Qm>
VADD<c><q>.<dt>  {<Dd>,} <Dn>, <Dm>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VADD` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

`I8`	size = 0b00
`I16`	size = 0b01
`I32`	size = 0b10
`I64`	size = 0b11

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] + Elem[D[m+r],e,esize];
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.272 VADD (floating-point)

Vector Add adds corresponding elements in two vectors, and places the results in the destination vector.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
VADD<c>.F32 <Qd>, <Qn>, <Qm>
VADD<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VADD<c>.F64 <Dd>, <Dn>, <Dm>
VADD<c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	1	1	Vn				Vd				1	0	1	sz	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	D	1	1	Vn				Vd				1	0	1	sz	N	0	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP vectors Encoding T2 / A2 can operate on VFP vectors under control of the FPSCR.LEN and FPSCR.STRIDE bits. For details see Appendix F *VFP Vector Operation Support*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.272 VADD（浮動小数点）

VADD（ベクタ加算（浮動小数点））は、2 つのベクタの対応するエレメントを加算し、結果をデスティネーションベクタに配置します。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VADD<c>.F32 <Qd>, <Qn>, <Qm>
VADD<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VADD<c>.F64 <Dd>, <Dn>, <Dm>
VADD<c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	1	1	Vn				Vd				1	0	1	sz	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	D	1	1	Vn				Vd				1	0	1	sz	N	0	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP ベクタ エンコード T2/A2 は、FPSCR.LEN ビットと FPSCR.STRIDE ビットの制御の下で VFP ベクタに作用することができます。詳細については、付録 F *「VFP ベクタ演算のサポート」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VADD<c><q>.F32 {<Qd>,} <Qn>, <Qm>`	Encoding T1 / A1, Q = 1, sz = 0
`VADD<c><q>.F32 {<Dd>,} <Dn>, <Dm>`	Encoding T1 / A1, Q = 0, sz = 0
`VADD<c><q>.F64 {<Dd>,} <Dn>, <Dm>`	Encoding T2 / A2, sz = 1
`VADD<c><q>.F32 {<Sd>,} <Sn>, <Sm>`	Encoding T2 / A2, sz = 0

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VADD` instruction must be unconditional.
`<Qd>, <Qn>, <Qm>`	The destination vector and the operand vectors, for a quadword operation.
`<Dd>, <Dn>, <Dm>`	The destination vector and the operand vectors, for a doubleword operation.
`<Sd>, <Sn>, <Sm>`	The destination vector and the operand vectors, for a singleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], FALSE);
    else            // VFP instruction
        if dp_operation then
            D[d] = FPAdd(D[n], D[m], TRUE);
        else
            S[d] = FPAdd(S[n], S[m], TRUE);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, Overflow, Underflow, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VADD<c><q>.F32 {<Qd>,} <Qn>, <Qm>`	エンコード T1/A1、Q = 1、sz = 0
`VADD<c><q>.F32 {<Dd>,} <Dn>, <Dm>`	エンコード T1/A1、Q = 0、sz = 0
`VADD<c><q>.F64 {<Dd>,} <Dn>, <Dm>`	エンコード T2/A2、sz = 1
`VADD<c><q>.F32 {<Sd>,} <Sn>, <Sm>`	エンコード T2/A2、sz = 0

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VADD` は常に無条件命令です。
`<Qd>, <Qn>, <Qm>`	クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Dd>, <Dn>, <Dm>`	ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Sd>, <Sn>, <Sm>`	シングルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], FALSE);
    else            // VFP instruction
        if dp_operation then
            D[d] = FPAdd(D[n], D[m], TRUE);
        else
            S[d] = FPAdd(S[n], S[m], TRUE);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、オーバフロー、アンダーフロー、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.273 VADDHN

Vector Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are truncated. (For rounded results, see *VRADDHN* on page A8-726).

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Encoding T1 / A1 Advanced SIMD

```
VADDHN<c>.<dt> <Dd>, <Qn>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	size		Vn				Vd				0	1	0	0	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	size		Vn				Vd				0	1	0	0	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.273 VADDHN

VADDHN（上位半分を返すベクタ加算および縮小）は、2 つのクワッドワードベクタの対応するエレメントを加算し、各結果の上位半分をダブルワードベクタに配置します。結果は切り捨てられます（丸められる結果については、*VRADDHN* P. A8-726 を参照して下さい）。

オペランドのエレメントには、16 ビット、32 ビット、または 64 ビットの整数を使用できます。整数には符号付きと符号なしの区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VADDHN<c>.<dt> <Dd>, <Qn>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	size		Vn				Vd				0	1	0	0	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	size		Vn				Vd				0	1	0	0	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VADDHN<c><q>.<dt>  <Dd>, <Qn>, <Qm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM `VADDHN` instruction must be unconditional.
`<dt>`	The data type for the elements of the operands. It must be one of: `I16` size = 0b00 `I32` size = 0b01 `I64` size = 0b10.
`<Dd>, <Qn>, <Qm>`	The destination vector, the first operand vector, and the second operand vector.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Elem[Q[n>>1],e,2*esize] + Elem[Q[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VADDHN<c><q>.<dt>  <Dd>, <Qn>, <Qm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VADDHN` は常に無条件命令です。
`<dt>`	オペランドのエレメントのデータ型。次のいずれかを使用できます。 `I16` size = 0b00 `I32` size = 0b01 `I64` size = 0b10
`<Dd>, <Qn>, <Qm>`	デスティネーションベクタ、最初のオペランドベクタ、および 2 番目のオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Elem[Q[n>>1],e,2*esize] + Elem[Q[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.274 VADDL, VADDW

VADDL (Vector Add Long) adds corresponding elements in two doubleword vectors, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of both operands.

VADDW (Vector Add Wide) adds corresponding elements in one quadword and one doubleword vector, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of the doubleword operand.

Encoding T1 / A1 Advanced SIMD

```
VADDL<c>.<dt> <Qd>, <Dn>, <Dm>
VADDW<c>.<dt> <Qd>, <Qn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				0	0	0	op	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				0	0	0	op	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw == (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.274 VADDL、VADDW

VADDL（ベクタロング加算）は、2 つのダブルワードベクタの対応するエレメントを加算し、結果をクワッドワードベクタに配置します。加算する前に、両方のオペランドのエレメントを符号拡張またはゼロ拡張します。

VADDW（ベクタワイド加算）は、1 つのクワッドワードベクタと 1 つのダブルワードベクタの対応するエレメントを加算し、結果をクワッドワードベクタに配置します。加算する前に、ダブルワードオペランドのエレメントを符号拡張またはゼロ拡張します。

エンコード T1/A1 アドバンスト SIMD

```
VADDL<c>.<dt> <Qd>, <Dn>, <Dm>
VADDW<c>.<dt> <Qd>, <Qn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				0	0	0	op	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				0	0	0	op	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw == (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VADDL<c><q>.<dt> <Qd>, <Dn>, <Dm>`	Encoded as op = 0
`VADDW<c><q>.<dt> {<Qd>,} <Qn>, <Dm>`	Encoded as op = 1

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VADDL` or `VADDW` instruction must be unconditional.

`<dt>` The data type for the elements of the second operand vector. It must be one of:

- `S8` encoded as size = 0b00, U = 0
- `S16` encoded as size = 0b01, U = 0
- `S32` encoded as size = 0b10, U = 0
- `U8` encoded as size = 0b00, U = 1
- `U16` encoded as size = 0b01, U = 1
- `U32` encoded as size = 0b10, U = 1.

`<Qd>` The destination register. If this register is omitted in a `VADDW` instruction, it is the same register as `<Qn>`.

`<Qn>, <Dm>` The first and second operand registers for a `VADDW` instruction.

`<Dn>, <Dm>` The first and second operand registers for a `VADDL` instruction.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vaddw then
            op1 = Int(Elem[Q[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[D[n],e,esize], unsigned);
        result = op1 + Int(Elem[D[m],e,esize],unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VADDL<c><q>.<dt> <Qd>, <Dn>, <Dm>`	op = 0 としてエンコードされる
`VADDW<c><q>.<dt> {<Qd>,} <Qn>, <Dm>`	op = 1 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VADDL` および `VADDW` は常に無条件命令です。

`<dt>` 2 番目のオペランドベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S8` size = 0b00、U = 0 としてエンコードされます。
- `S16` size = 0b01、U = 0 としてエンコードされます。
- `S32` size = 0b10、U = 0 としてエンコードされます。
- `U8` size = 0b00、U = 1 としてエンコードされます。
- `U16` size = 0b01、U = 1 としてエンコードされます。
- `U32` size = 0b10、U = 1 としてエンコードされます。

`<Qd>` デスティネーションレジスタ。`VADDW` 命令でこのレジスタを省略した場合、`<Qn>` と同じレジスタになります。

`<Qn>, <Dm>` `VADDW` 命令の最初および 2 番目のオペランドレジスタ

`<Dn>, <Dm>` `VADDL` 命令の最初および 2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vaddw then
            op1 = Int(Elem[Q[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[D[n],e,esize], unsigned);
        result = op1 + Int(Elem[D[m],e,esize],unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.275 VAND (immediate)

This is a pseudo-instruction, equivalent to a `VBIC` (immediate) instruction with the immediate value bitwise inverted. For details see *VBIC (immediate)* on page A8-546.

A8.6.276 VAND (register)

This instruction performs a bitwise AND operation between two registers, and places the result in the destination register.

Encoding T1 / A1 Advanced SIMD

```
VAND<c> <Qd>, <Qn>, <Qm>
VAND<c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.275 VAND（イミディエート）

VAND（イミディエート）は擬似命令で、イミディエート値がビット単位で反転された `VBIC`（イミディエート）命令と等価です。詳細については、*VBIC（イミディエート）*P. A8-546 を参照して下さい。

A8.6.276 VAND（レジスタ）

VAND（レジスタ）命令は、2 つのレジスタの間でビット単位の論理積演算を実行し、結果をデスティネーションレジスタに配置します。

エンコード T1/A1 アドバンスト SIMD

```
VAND<c> <Qd>, <Qn>, <Qm>
VAND<c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VAND<c><q>{.<dt>}  {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VAND<c><q>{.<dt>}  {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM `VAND` instruction must be unconditional.
`<dt>`	An optional data type. It is ignored by assemblers, and does not affect the encoding.
`<Qd>, <Qn>, <Qm>`	The destination vector and the operand vectors, for a quadword operation.
`<Dd>, <Dn>, <Dm>`	The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND D[m+r];
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VAND<c><q>{.<dt>}  {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VAND<c><q>{.<dt>}  {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VAND` は常に無条件命令です。
`<dt>`	オプションのデータ型。アセンブラによって無視され、エンコードに影響を与えません。
`<Qd>, <Qn>, <Qm>`	クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Dd>, <Dn>, <Dm>`	ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND D[m+r];
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.277 VBIC (immediate)

Vector Bitwise Bit Clear (immediate) performs a bitwise AND between a register value and the complement of an immediate value, and returns the result into the destination vector. For the range of constants available, see *One register and a modified immediate value* on page A7-21.

Encoding T1 / A1 Advanced SIMD

```
VBIC<c>.<dt> <Qd>, #<imm>
VBIC<c>.<dt> <Dd>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd				cmode				0	Q	1	1	imm4			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd				cmode				0	Q	1	1	imm4			

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.277 VBIC（イミディエート）

VBIC（ベクタビット単位ビットクリア（イミディエート））は、レジスタの値とイミディエート値の補数との間でビット単位の論理積を実行し、結果をデスティネーションベクタに返します。使用可能な定数の範囲については、*1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VBIC<c>.<dt> <Qd>, #<imm>
VBIC<c>.<dt> <Dd>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd				cmode				0	Q	1	1	imm4			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd				cmode				0	Q	1	1	imm4			

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VBIC<c><q>.<dt>  {<Qd>,} <Qd>, #<imm>        Encoded as Q = 1
VBIC<c><q>.<dt>  {<Dd>,} <Dd>, #<imm>>       Encoded as Q = 0
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VBIC` instruction must be unconditional.

`<dt>` The data type used for `<imm>`. It can be either `I16` or `I32`.

`I8`, `I64`, and `F32` are also permitted, but the resulting syntax is a pseudo-instruction.

`<Qd>` The destination vector for a quadword operation.

`<Dd>` The destination vector for a doubleword operation.

`<imm>` A constant of the type specified by `<dt>`. This constant is replicated enough times to fill the destination register. For example, `VBIC.I32 D0,#10` ANDs the complement of `0x0000000A0000000A` with D0, and puts the result into D0.

For details of the range of constants available and the encoding of `<dt>` and `<imm>`, see *One register and a modified immediate value* on page A7-21.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[d+r] AND NOT(imm64);
```

Exceptions

Undefined Instruction.

Pseudo-instructions

`VAND` can be used with a range of constants that are the bitwise inverse of the available constants for `VBIC`. This is assembled as the equivalent `VBIC` instruction. Disassembly produces the `VBIC` form.

One register and a modified immediate value on page A7-21 describes pseudo-instructions with a combination of `<dt>` and `<imm>` that is not supported by hardware, but that generates the same destination register value as a different combination that is supported by hardware.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VBIC<c><q>.<dt>  {<Qd>,} <Qd>, #<imm>        Q = 1 としてエンコードされる
VBIC<c><q>.<dt>  {<Dd>,} <Dd>, #<imm>>       Q = 0 としてエンコードされる
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VBIC` は常に無条件命令です。

`<dt>` `<imm>` で使用されるデータ型。`I16` または `I32` を指定できます。

`I8`、`I64`、および `F32` も指定できますが、指定した場合、構文が擬似命令になります。

`<Qd>` クワッドワード演算の場合のデスティネーションベクタ

`<Dd>` ダブルワード演算の場合のデスティネーションベクタ

`<imm>` `<dt>` で指定されたデータ型の定数。この定数は、デスティネーションレジスタを満たすのに必要な回数だけ複製されます。例えば、`VBIC.I32 D0,#10` では、`0x0000000A0000000A` の補数と D0 との間で論理積が実行され、結果が D0 に配置されます。

使用可能な定数の範囲と、`<dt>` および `<imm>` のエンコードの詳細については、*1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[d+r] AND NOT(imm64);
```

例外

未定義命令

擬似命令

`VAND` は、`VBIC` で使用可能な定数のビット単位の反転である定数の範囲で使用することができます。この命令は、等価な `VBIC` 命令としてアセンブルされます。逆アセンブルでは、`VBIC` 形式が生成されます。

ハードウェアでサポートされていないにもかかわらず、ハードウェアでサポートされている別の組み合わせと同じデスティネーションレジスタの値を生成する、`<dt>` と `<imm>` の組み合わせを持つ擬似命令については、*1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.278 VBIC (register)

Vector Bitwise Bit Clear (register) performs a bitwise AND between a register value and the complement of a register value, and places the result in the destination register.

Encoding T1 / A1 Advanced SIMD

```
VBIC<c> <Qd>, <Qn>, <Qm>
VBIC<c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.278 VBIC（レジスタ）

VBIC（ベクタビット単位ビットクリア（レジスタ））は、レジスタの値とレジスタの値の補数との間でビット単位の論理積を実行し、結果をデスティネーションベクタに配置します。

エンコード T1/A1 アドバンスト SIMD

```
VBIC<c> <Qd>, <Qn>, <Qm>
VBIC<c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VBIC<c><q>{.<dt>}  {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VBIC<c><q>{.<dt>}  {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM `VBIC` instruction must be unconditional.
`<dt>`	An optional data type. It is ignored by assemblers, and does not affect the encoding.
`<Qd>, <Qn>, <Qm>`	The destination vector and the operand vectors, for a quadword operation.
`<Dd>, <Dn>, <Dm>`	The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND NOT(D[m+r]);
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VBIC<c><q>{.<dt>}  {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VBIC<c><q>{.<dt>}  {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VBIC` は常に無条件命令です。
`<dt>`	オプションのデータ型。アセンブラによって無視され、エンコードに影響を与えません。
`<Qd>, <Qn>, <Qm>`	クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Dd>, <Dn>, <Dm>`	ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND NOT(D[m+r]);
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.279 VBIF, VBIT, VBSL

VBIF (Vector Bitwise Insert if False), VBIT (Vector Bitwise Insert if True), and VBSL (Vector Bitwise Select) perform bitwise selection under the control of a mask, and place the results in the destination register. The registers can be either quadword or doubleword, and must all be the same size.

Encoding T1 / A1 Advanced SIMD

```
V<op><c> <Qd>, <Qn>, <Qm>
V<op><c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	op		Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	op		Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE VEOR;
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.279 VBIF、VBIT、VBSL

VBIF（FALSE 時ベクタビット単位挿入）、VBIT（TRUE 時ベクタビット単位挿入）、および VBSL（ベクタビット単位選択）は、マスクの制御の下でビット単位の選択を実行し、結果をデスティネーションレジスタに配置します。これらのレジスタはクワッドワードまたはダブルワードで、すべて同じサイズの必要があります。

エンコード T1/A1 アドバンスト SIMD

```
V<op><c> <Qd>, <Qn>, <Qm>
V<op><c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	op		Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	op		Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE VEOR;
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
V<op><c><q>{.<dt>}  {<Qd>,} <Qn>, <Qm>        Encoded as Q = 1
V<op><c><q>{.<dt>}  {<Dd>,} <Dn>, <Dm>        Encoded as Q = 0
```

where:

`<op>` The operation. It must be one of:

- `BIF` Bitwise Insert if False, encoded as op = 0b11. Inserts each bit from Vn into Vd if the corresponding bit of Vm is 0, otherwise leaves the Vd bit unchanged.
- `BIT` Bitwise Insert if True, encoded as op = 0b10. Inserts each bit from Vn into Vd if the corresponding bit of Vm is 1, otherwise leaves the Vd bit unchanged.
- `BSL` Bitwise Select, encoded as op = 0b01. Selects each bit from Vn into Vd if the corresponding bit of Vd is 1, otherwise selects the bit from Vm.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VBIF`, `VBIT`, or `VBSL` instruction must be unconditional.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]);
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]);
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]);
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
V<op><c><q>{.<dt>}  {<Qd>,} <Qn>, <Qm>        Q = 1 としてエンコードされる
V<op><c><q>{.<dt>}  {<Dd>,} <Dn>, <Dm>        Q = 0 としてエンコードされる
```

各項目の説明については以下を参照して下さい。

`<op>` 演算。次のいずれかを使用できます。

- `BIF` FALSE 時のビット単位挿入で、op = 0b11 としてエンコードされます。Vm の対応するビットが 0 の場合は、Vn から各ビットを Vd に挿入し、それ以外の場合は Vd のビットを変更しません。
- `BIT` TRUE 時のビット単位挿入で、op = 0b10 としてエンコードされます。Vm の対応するビットが 1 の場合は、Vn から各ビットを Vd に挿入し、それ以外の場合は Vd のビットを変更しません。
- `BSL` ビット単位選択で、op = 0b01 としてエンコードされます。Vd の対応するビットが 1 の場合は、Vn の各ビットを選択して Vd に挿入し、それ以外の場合は Vm からのビットを選択します。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VBIF`、`VBIT`、`VBSL` は常に無条件命令です。

`<dt>` オプションのデータ型。アセンブラによって無視され、エンコードに影響を与えません。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]);
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]);
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]);
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.280 VCEQ (register)

VCEQ (Vector Compare Equal) takes each element in a vector, and compares it with the corresponding element of a second vector. If they are equal, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.
- 32-bit floating-point numbers.

The result vector elements are bitfields the same size as the operand vector elements.

Encoding T1 / A1 Advanced SIMD

```
VCEQ<c>.<dt> <Qd>, <Qn>, <Qm>          <dt> an integer type
VCEQ<c>.<dt> <Dd>, <Dn>, <Dm>          <dt> an integer type
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
int_operation = TRUE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 Advanced SIMD (UNDEFINED in integer-only variant)

```
VCEQ<c>.F32 <Qd>, <Qn>, <Qm>
VCEQ<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
int_operation = FALSE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.280 VCEQ（レジスタ）

VCEQ（ベクタ比較等価）は、ベクタの各エレメントを取得して、2 番目のベクタの対応するエレメントと比較します。それらが等しい場合、デスティネーションベクタの対応するエレメントがすべて 1 にセットされます。それ以外の場合、すべて 0 にセットされます。

オペランドベクタのエレメントには、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットの整数。整数には符号付きと符号なしの区別はありません。
- 32 ビットの浮動小数点数

結果ベクタのエレメントはビットフィールドで、オペランドベクタのエレメントと同じサイズです。

エンコード T1/A1 アドバンスト SIMD

```
VCEQ<c>.<dt> <Qd>, <Qn>, <Qm>          <dt> は整数型
VCEQ<c>.<dt> <Dd>, <Dn>, <Dm>          <dt> は整数型
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
int_operation = TRUE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VCEQ<c>.F32 <Qd>, <Qn>, <Qm>
VCEQ<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
int_operation = FALSE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCEQ<c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VCEQ<c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VCEQ` instruction must be unconditional.

`<dt>` The data types for the elements of the operands. It must be one of:

- `I8` encoding T1 / A1, size = 0b00
- `I16` encoding T1 / A1, size = 0b01
- `I32` encoding T1 / A1, size = 0b10
- `F32` encoding T2 / A2, sz = 0.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            if int_operation then
                test_passed = (op1 == op2);
            else
                test_passed = FPCompareEQ(op1, op2, FALSE);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal and Invalid Operation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`VCEQ<c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VCEQ<c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VCEQ` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `I8` エンコード T1/A1、size = 0b00
- `I16` エンコード T1/A1、size = 0b01
- `I32` エンコード T1/A1、size = 0b10
- `F32` エンコード T2/A2、sz = 0

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            if int_operation then
                test_passed = (op1 == op2);
            else
                test_passed = FPCompareEQ(op1, op2, FALSE);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.281 VCEQ (immediate #0)

`VCEQ #0` (Vector Compare Equal to zero) takes each element in a vector, and compares it with zero. If it is equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.
- 32-bit floating-point numbers.

The result vector elements are bitfields the same size as the operand vector elements.

Encoding T1 / A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

```
VCEQ<c>.<dt> <Qd>, <Qm>, #0
VCEQ<c>.<dt> <Dd>, <Dm>, #0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	0	1	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	0	1	0	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.281 VCEQ（イミディエート #0）

`VCEQ #0`（ベクタ比較 0 に等価）は、ベクタの各エレメントを取得して、0 と比較します。0 に等しい場合、デスティネーションベクタの対応するエレメントがすべて 1 にセットされます。それ以外の場合、すべて 0 にセットされます。

オペランドベクタのエレメントには、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットの整数。整数には符号付きと符号なしの区別はありません。
- 32 ビットの浮動小数点数

結果ベクタのエレメントはビットフィールドで、オペランドベクタのエレメントと同じサイズです。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは F = 1 は未定義）

```
VCEQ<c>.<dt> <Qd>, <Qm>, #0
VCEQ<c>.<dt> <Dd>, <Dm>, #0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	0	1	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	0	1	0	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VCEQ<c><q>.<dt>  {<Qd>,} <Qm>, #0          Encoded as Q = 1
VCEQ<c><q>.<dt>  {<Dd>,} <Dm>, #0          Encoded as Q = 0
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VCEQ` instruction must be unconditional.

`<dt>` The data types for the elements of the operands. It must be one of:

- `I8` encoded as size = 0b00, F = 0
- `I16` encoded as size = 0b01, F = 0
- `I32` encoded as size = 0b10, F = 0
- `F32` encoded as size = 0b10, F = 1.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                test_passed = FPCompareEQ(Elem[D[m+r],e,esize], FPZero('0',esize), FALSE);
            else
                test_passed = (Elem[D[m+r],e,esize] == Zeros(esize));
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal and Invalid Operation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VCEQ<c><q>.<dt>  {<Qd>,} <Qm>, #0          Q = 1 としてエンコードされる
VCEQ<c><q>.<dt>  {<Dd>,} <Dm>, #0          Q = 0 としてエンコードされる
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VCEQ` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `I8` size = 0b00、F = 0 としてエンコードされます。
- `I16` size = 0b01、F = 0 としてエンコードされます。
- `I32` size = 0b10、F = 0 としてエンコードされます。
- `F32` size = 0b10、F = 1 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                test_passed = FPCompareEQ(Elem[D[m+r],e,esize], FPZero('0',esize), FALSE);
            else
                test_passed = (Elem[D[m+r],e,esize] == Zeros(esize));
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.282 VCGE (register)

VCGE (Vector Compare Greater Than or Equal) takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers
- 8-bit, 16-bit, or 32-bit unsigned integers
- 32-bit floating-point numbers.

The result vector elements are bitfields the same size as the operand vector elements.

Encoding T1 / A1 Advanced SIMD

```
VCGE<c>.<dt> <Qd>, <Qn>, <Qm>          <dt> an integer type
VCGE<c>.<dt> <Dd>, <Dn>, <Dm>          <dt> an integer type
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	1	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	1	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
type = if U == '1' then VCGEtype_unsigned else VCGEtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 Advanced SIMD (UNDEFINED in integer-only variant)

```
VCGE<c>.F32 <Qd>, <Qn>, <Qm>
VCGE<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
type = VCGEtype_fp;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.282 VCGE（レジスタ）

VCGE（ベクタ比較以上）は、ベクタの各エレメントを取得して、2 番目のベクタの対応するエレメントと比較します。最初のベクタのエレメントが 2 番目のベクタのものより大きいか等しい場合、デスティネーションベクタの対応する要素がすべて 1 にセットされます。それ以外の場合、すべて 0 にセットされます。

オペランドベクタのエレメントには、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットの符号付き整数
- 8 ビット、16 ビット、または 32 ビットの符号なし整数
- 32 ビットの浮動小数点数

結果ベクタのエレメントはビットフィールドで、オペランドベクタのエレメントと同じサイズです。

エンコード T1/A1 アドバンスト SIMD

```
VCGE<c>.<dt> <Qd>, <Qn>, <Qm>          <dt> は整数型
VCGE<c>.<dt> <Dd>, <Dn>, <Dm>          <dt> は整数型
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	1	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	1	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
type = if U == '1' then VCGEtype_unsigned else VCGEtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VCGE<c>.F32 <Qd>, <Qn>, <Qm>
VCGE<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
type = VCGEtype_fp;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCGE<c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VCGE<c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VCGE` instruction must be unconditional.

`<dt>` The data types for the elements of the operands. It must be one of:

- `S8` encoding T1 / A1, size = 0b00, U = 0
- `S16` encoding T1 / A1, size = 0b01, U = 0
- `S32` encoding T1 / A1, size = 0b10, U = 0
- `U8` encoding T1 / A1, size = 0b00, U = 1
- `U16` encoding T1 / A1, size = 0b01, U = 1
- `U32` encoding T1 / A1, size = 0b10, U = 1
- `F32` encoding T2 / A2, sz = 0.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
enumeration VCGEtype {VCGEtype_signed, VCGEtype_unsigned, VCGEtype_fp};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            case type of
                when VCGEtype_signed    test_passed = (SInt(op1) >= SInt(op2));
                when VCGEtype_unsigned  test_passed = (UInt(op1) >= UInt(op2));
                when VCGEtype_fp        test_passed = FPCompareGE(op1, op2, FALSE);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal and Invalid Operation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCGE<c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VCGE<c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VCGE` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S8` エンコード T1/A1、size = 0b00、U = 0
- `S16` エンコード T1/A1、size = 0b01、U = 0
- `S32` エンコード T1/A1、size = 0b10、U = 0
- `U8` エンコード T1/A1、size = 0b00、U = 1
- `U16` エンコード T1/A1、size = 0b01、U = 1
- `U32` エンコード T1/A1、size = 0b10、U = 1
- `F32` エンコード T2/A2、sz = 0

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
enumeration VCGEtype {VCGEtype_signed, VCGEtype_unsigned, VCGEtype_fp};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            case type of
                when VCGEtype_signed    test_passed = (SInt(op1) >= SInt(op2));
                when VCGEtype_unsigned  test_passed = (UInt(op1) >= UInt(op2));
                when VCGEtype_fp        test_passed = FPCompareGE(op1, op2, FALSE);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.283 VCGE (immediate #0)

`VCGE #0` (Vector Compare Greater Than or Equal to Zero) take each element in a vector, and compares it with zero. If it is greater than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers
- 32-bit floating-point numbers.

The result vector elements are bitfields the same size as the operand vector elements.

Encoding T1 / A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

```
VCGE<c>.<dt> <Qd>, <Qm>, #0
VCGE<c>.<dt> <Dd>, <Dm>, #0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	0	0	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	0	0	1	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.283 VCGE（イミディエート #0）

`VCGE #0`（ベクタ比較 0 以上）は、ベクタの各エレメントを取得して、0 と比較します。0 以上の場合、デスティネーションベクタの対応するエレメントがすべて 1 にセットされます。それ以外の場合、すべて 0 にセットされます。

オペランドベクタのエレメントには、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットの符号付き整数
- 32 ビットの浮動小数点数

結果ベクタのエレメントはビットフィールドで、オペランドベクタのエレメントと同じサイズです。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは F = 1 は未定義）

```
VCGE<c>.<dt> <Qd>, <Qm>, #0
VCGE<c>.<dt> <Dd>, <Dm>, #0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	0	0	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	0	0	1	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCGE<c><q>.<dt> {<Qd>,} <Qm>, #0`	Encoded as Q = 1
`VCGE<c><q>.<dt> {<Dd>,} <Dm>, #0`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VCGE` instruction must be unconditional.

`<dt>` The data types for the elements of the operands. It must be one of:

- `S8` encoded as size = 0b00, F = 0
- `S16` encoded as size = 0b01, F = 0
- `S32` encoded as size = 0b10, F = 0
- `F32` encoded as size = 0b10, F = 1.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                test_passed = FPCompareGE(Elem[D[m+r],e,esize], FPZero('0',esize), FALSE);
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) >= 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal and Invalid Operation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCGE<c><q>.<dt> {<Qd>,} <Qm>, #0`	Q = 1 としてエンコードされる
`VCGE<c><q>.<dt> {<Dd>,} <Dm>, #0`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VCGE` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S8` size = 0b00、F = 0 としてエンコードされます。
- `S16` size = 0b01、F = 0 としてエンコードされます。
- `S32` size = 0b10、F = 0 としてエンコードされます。
- `F32` size = 0b10、F = 1 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                test_passed = FPCompareGE(Elem[D[m+r],e,esize], FPZero('0',esize), FALSE);
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) >= 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.284 VCGT (register)

VCGT (Vector Compare Greater Than) takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers
- 8-bit, 16-bit, or 32-bit unsigned integers
- 32-bit floating-point numbers.

The result vector elements are bitfields the same size as the operand vector elements.

Encoding T1 / A1 Advanced SIMD

```
VCGT<c>.<dt> <Qd>, <Qn>, <Qm>          <dt> an integer type
VCGT<c>.<dt> <Dd>, <Dn>, <Dm>          <dt> an integer type
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	1	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	1	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
type = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 Advanced SIMD (UNDEFINED in integer-only variant)

```
VCGT<c>.F32 <Qd>, <Qn>, <Qm>
VCGT<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
type = VCGTtype_fp;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.284 VCGT（レジスタ）

VCGT（ベクタ比較より大きい（レジスタ））は、ベクタの各エレメントを取得して、2 番目のベクタの対応するエレメントと比較します。2 番目のエレメントより大きい場合、デスティネーションベクタの対応するエレメントがすべて 1 にセットされます。それ以外の場合、すべて 0 にセットされます。

オペランドベクタのエレメントには、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットの符号付き整数
- 8 ビット、16 ビット、または 32 ビットの符号なし整数
- 32 ビットの浮動小数点数

結果ベクタのエレメントはビットフィールドで、オペランドベクタのエレメントと同じサイズです。

エンコード T1/A1 アドバンスト SIMD

```
VCGT<c>.<dt> <Qd>, <Qn>, <Qm>          <dt> は整数型
VCGT<c>.<dt> <Dd>, <Dn>, <Dm>          <dt> は整数型
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	1	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	1	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
type = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VCGT<c>.F32 <Qd>, <Qn>, <Qm>
VCGT<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
type = VCGTtype_fp;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCGT<c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VCGT<c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VCGT` instruction must be unconditional.

`<dt>` The data types for the elements of the operands. It must be one of:

`S8`	encoding T1 / A1, size = 0b00, U = 0
`S16`	encoding T1 / A1, size = 0b01, U = 0
`S32`	encoding T1 / A1, size = 0b10, U = 0
`U8`	encoding T1 / A1, size = 0b00, U = 1
`U16`	encoding T1 / A1, size = 0b01, U = 1
`U32`	encoding T1 / A1, size = 0b10, U = 1
`F32`	encoding T2 / A2, sz = 0.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            case type of
                when VCGTtype_signed    test_passed = (SInt(op1) > SInt(op2));
                when VCGTtype_unsigned  test_passed = (UInt(op1) > UInt(op2));
                when VCGTtype_fp        test_passed = FPCompareGT(op1, op2, FALSE);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal and Invalid Operation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCGT<c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VCGT<c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VCGT` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

`S8`	エンコード T1/A1、size = 0b00、U = 0
`S16`	エンコード T1/A1、size = 0b01、U = 0
`S32`	エンコード T1/A1、size = 0b10、U = 0
`U8`	エンコード T1/A1、size = 0b00、U = 1
`U16`	エンコード T1/A1、size = 0b01、U = 1
`U32`	エンコード T1/A1、size = 0b10、U = 1
`F32`	エンコード T2/A2、sz = 0

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            case type of
                when VCGTtype_signed    test_passed = (SInt(op1) > SInt(op2));
                when VCGTtype_unsigned  test_passed = (UInt(op1) > UInt(op2));
                when VCGTtype_fp        test_passed = FPCompareGT(op1, op2, FALSE);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.285 VCGT (immediate #0)

`VCGT #0` (Vector Compare Greater Than Zero) take each element in a vector, and compares it with zero. If it is greater than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers
- 32-bit floating-point numbers.

The result vector elements are bitfields the same size as the operand vector elements.

Encoding T1 / A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

```
VCGT<c>.<dt> <Qd>, <Qm>, #0
VCGT<c>.<dt> <Dd>, <Dm>, #0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	0	0	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	0	0	0	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.285 VCGT（イミディエート #0）

`VCGT #0`（ベクタ比較 0 より大きい）は、ベクタの各エレメントを取得して、0 と比較します。0 より大きい場合、デスティネーションベクタの対応するエレメントがすべて 1 にセットされます。それ以外の場合、すべて 0 にセットされます。

オペランドベクタのエレメントには、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットの符号付き整数
- 32 ビットの浮動小数点数

結果ベクタのエレメントはビットフィールドで、オペランドベクタのエレメントと同じサイズです。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは F = 1 は未定義）

```
VCGT<c>.<dt> <Qd>, <Qm>, #0
VCGT<c>.<dt> <Dd>, <Dm>, #0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	0	0	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	0	0	0	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCGT<c><q>.<dt> {<Qd>,} <Qm>, #0`	Encoded as Q = 1
`VCGT<c><q>.<dt> {<Dd>,} <Dm>, #0`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VCGT` instruction must be unconditional.

`<dt>` The data types for the elements of the operands. It must be one of:

- `S8` encoded as size = 0b00, F = 0
- `S16` encoded as size = 0b01, F = 0
- `S32` encoded as size = 0b10, F = 0
- `F32` encoded as size = 0b10, F = 1.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                test_passed = FPCompareGT(Elem[D[m+r],e,esize], FPZero('0',esize), FALSE);
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) > 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal and Invalid Operation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCGT<c><q>.<dt> {<Qd>,} <Qm>, #0`	Q = 1 としてエンコードされる
`VCGT<c><q>.<dt> {<Dd>,} <Dm>, #0`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VCGT` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S8` size = 0b00、F = 0 としてエンコードされます。
- `S16` size = 0b01、F = 0 としてエンコードされます。
- `S32` size = 0b10、F = 0 としてエンコードされます。
- `F32` size = 0b10、F = 1 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                test_passed = FPCompareGT(Elem[D[m+r],e,esize], FPZero('0',esize), FALSE);
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) > 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.286 VCLE (register)

VCLE is a pseudo-instruction, equivalent to a VCGE instruction with the operands reversed. For details see *VCGE (register)* on page A8-556.

A8.6.287 VCLE (immediate #0)

VCLE #0 (Vector Compare Less Than or Equal to Zero) take each element in a vector, and compares it with zero. If it is less than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers
- 32-bit floating-point numbers.

The result vector elements are bitfields the same size as the operand vector elements.

Encoding T1 / A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

```
VCLE<c>.<dt> <Qd>, <Qm>, #0
VCLE<c>.<dt> <Dd>, <Dm>, #0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	0	1	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	0	1	1	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.286 VCLE（レジスタ）

VCLE は擬似命令で、オペランドが反転された VCGE 命令と等価です。詳細については、*VCGE（レジスタ）* P. A8- 556 を参照して下さい。

A8.6.287 VCLE（イミディエート #0）

VCLE #0（ベクタ比較 0 以下）は、ベクタの各エレメントを取得して、0 と比較します。0 以下の場合、デスティネーションベクタの対応するエレメントがすべて 1 にセットされます。それ以外の場合、すべて 0 にセットされます。

オペランドベクタのエレメントには、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットの符号付き整数
- 32 ビットの浮動小数点数

結果ベクタのエレメントはビットフィールドで、オペランドベクタのエレメントと同じサイズです。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは F = 1 は未定義）

```
VCLE<c>.<dt> <Qd>, <Qm>, #0
VCLE<c>.<dt> <Dd>, <Dm>, #0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	0	1	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	0	1	1	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCLE<c><q>.<dt> {<Qd>,} <Qm>, #0`	Encoded as Q = 1
`VCLE<c><q>.<dt> {<Dd>,} <Dm>, #0`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VCLE` instruction must be unconditional.

`<dt>` The data types for the elements of the operands. It must be one of:

- `S8` encoded as size = 0b00, F = 0
- `S16` encoded as size = 0b01, F = 0
- `S32` encoded as size = 0b10, F = 0
- `F32` encoded as size = 0b10, F = 1.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                test_passed = FPCompareGE(FPZero('0',esize), Elem[D[m+r],e,esize], FALSE);
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) <= 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal and Invalid Operation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCLE<c><q>.<dt> {<Qd>,} <Qm>, #0`	Q = 1 としてエンコードされる
`VCLE<c><q>.<dt> {<Dd>,} <Dm>, #0`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VCLE` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S8` size = 0b00、F = 0 としてエンコードされます。
- `S16` size = 0b01、F = 0 としてエンコードされます。
- `S32` size = 0b10、F = 0 としてエンコードされます。
- `F32` size = 0b10、F = 1 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                test_passed = FPCompareGE(FPZero('0',esize), Elem[D[m+r],e,esize], FALSE);
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) <= 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.288 VCLS

Vector Count Leading Sign Bits counts the number of consecutive bits following the topmost bit, that are the same as the topmost bit, in each element in a vector, and places the results in a second vector. The count does not include the topmost bit itself.

The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit signed integers.

The result vector elements are the same data type as the operand vector elements.

Encoding T1 / A1 Advanced SIMD

```
VCLS<c>.<dt> <Qd>, <Qm>
VCLS<c>.<dt> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	0	0	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	0	0	0	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.288 VCLS

VCLS（ベクタ先頭符号ビットカウント）は、ベクタの各エレメントの最上位ビットに続く連続ビット（最上位ビットと同じ）の数をカウントし、結果を 2 番目のベクタに配置します。最上位ビット自体はカウントに含まれません。

オペランドベクタのエレメントには、8 ビット、16 ビット、または 32 ビットの符号付き整数を使用できます。

結果ベクタのエレメントのデータ型は、オペランドベクタのエレメントと同じです。

エンコード T1/A1 アドバンスト SIMD

```
VCLS<c>.<dt> <Qd>, <Qm>
VCLS<c>.<dt> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	0	0	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	0	0	0	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCLS<c><q>.<dt> <Qd>, <Qm>`	Encoded as Q = 1
`VCLS<c><q>.<dt> <Dd>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VCLS` instruction must be unconditional.

`<dt>` The data size for the elements of the operands. It must be one of:

- `S8` encoded as size = 0b00
- `S16` encoded as size = 0b01
- `S32` encoded as size = 0b10.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingSignBits(Elem[D[m+r],e,esize]);
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCLS<c><q>.<dt> <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VCLS<c><q>.<dt> <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VCLS` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータサイズ。次のいずれかを使用できます。

- `S8` size = 0b00 としてエンコードされます。
- `S16` size = 0b01 としてエンコードされます。
- `S32` size = 0b10 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingSignBits(Elem[D[m+r],e,esize]);
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.289 VCLT (register)

VCLT is a pseudo-instruction, equivalent to a VCGT instruction with the operands reversed. For details see *VCGT (register)* on page A8-560.

A8.6.290 VCLT (immediate #0)

VCLT #0 (Vector Compare Less Than Zero) take each element in a vector, and compares it with zero. If it is less than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers
- 32-bit floating-point numbers.

The result vector elements are bitfields the same size as the operand vector elements.

Encoding T1 / A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

```
VCLT<c>.<dt> <Qd>, <Qm>, #0
VCLT<c>.<dt> <Dd>, <Dm>, #0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	1	0	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	1	0	0	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.289 VCLT（レジスタ）

VCLT は擬似命令で、オペランドが反転された VCGT 命令と等価です。詳細については、*VCGT（レジスタ）* P. A8- 560 を参照して下さい。

A8.6.290 VCLT（イミディエート #0）

VCLT #0（ベクタ比較 0 未満）は、ベクタの各エレメントを取得して、0 と比較します。0 より小さい場合、デスティネーションベクタの対応するエレメントがすべて 1 にセットされます。それ以外の場合、すべて 0 にセットされます。

オペランドベクタのエレメントには、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットの符号付き整数
- 32 ビットの浮動小数点数

結果ベクタのエレメントはビットフィールドで、オペランドベクタのエレメントと同じサイズです。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは F = 1 は未定義）

```
VCLT<c>.<dt> <Qd>, <Qm>, #0
VCLT<c>.<dt> <Dd>, <Dm>, #0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	1	0	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	1	0	0	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCLT<c><q>.<dt> {<Qd>,} <Qm>, #0`	Encoded as Q = 1
`VCLT<c><q>.<dt> {<Dd>,} <Dm>, #0`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VCLT` instruction must be unconditional.

`<dt>` The data types for the elements of the operands. It must be one of:

- `S8` encoded as size = 0b00, F = 0
- `S16` encoded as size = 0b01, F = 0
- `S32` encoded as size = 0b10, F = 0
- `F32` encoded as size = 0b10, F = 1.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                test_passed = FPCompareGT(FPZero('0',esize), Elem[D[m+r],e,esize], FALSE);
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) < 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal and Invalid Operation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCLT<c><q>.<dt> {<Qd>,} <Qm>, #0`	Q = 1 としてエンコードされる
`VCLT<c><q>.<dt> {<Dd>,} <Dm>, #0`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VCLT` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S8` size = 0b00、F = 0 としてエンコードされます。
- `S16` size = 0b01、F = 0 としてエンコードされます。
- `S32` size = 0b10、F = 0 としてエンコードされます。
- `F32` size = 0b10、F = 1 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                test_passed = FPCompareGT(FPZero('0',esize), Elem[D[m+r],e,esize], FALSE);
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) < 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.291 VCLZ

Vector Count Leading Zeros counts the number of consecutive zeros, starting from the most significant bit, in each element in a vector, and places the results in a second vector.

The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The result vector elements are the same data type as the operand vector elements.

Encoding T1 / A1 Advanced SIMD

```
VCLZ<c>.<dt> <Qd>, <Qm>
VCLZ<c>.<dt> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	0	0	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	0	0	1	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.291 VCLZ

VCLZ（ベクタ先頭ゼロカウント）は、ベクタの各エレメントの連続する 0（最上位ビットから始まる）の数をカウントし、結果を 2 番目のベクタに配置します。

オペランドベクタのエレメントには、8 ビット、16 ビット、または 32 ビットの整数を使用できます。整数には符号付きと符号なしの区別はありません。

結果ベクタのエレメントのデータ型は、オペランドベクタのエレメントと同じです。

エンコード T1/A1 アドバンスト SIMD

```
VCLZ<c>.<dt> <Qd>, <Qm>
VCLZ<c>.<dt> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	0	0	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	0	0	1	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCLZ<c><q>.<dt> <Qd>, <Qm>`	Encoded as Q = 1
`VCLZ<c><q>.<dt> <Dd>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VCLZ` instruction must be unconditional.

`<dt>` The data size for the elements of the operands. It must be one of:

- `I8` encoded as size = 0b00
- `I16` encoded as size = 0b01
- `I32` encoded as size = 0b10.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingZeroBits(Elem[D[m+r],e,esize]);
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCLZ<c><q>.<dt> <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VCLZ<c><q>.<dt> <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VCLZ` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータサイズ。次のいずれかを使用できます。

- `I8` size = 0b00 としてエンコードされます。
- `I16` size = 0b01 としてエンコードされます。
- `I32` size = 0b10 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingZeroBits(Elem[D[m+r],e,esize]);
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.292 VCMP, VCMPE

This instruction compares two floating-point registers, or one floating-point register and zero. It writes the result to the FPSCR flags. These are normally transferred to the ARM flags by a subsequent VMRS instruction.

It can optionally raise an Invalid Operation exception if either operand is any type of NaN. It always raises an Invalid Operation exception if either operand is a signaling NaN.

Encoding T1 / A1 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VCMP{E}<c>.F64 <Dd>, <Dm>
VCMP{E}<c>.F32 <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	0	Vd				1	0	1	sz	E	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	1	0	0	Vd				1	0	1	sz	E	1	M	0	Vm			

```
dp_operation = (sz == '1');  quiet_nan_exc = (E == '1');  with_zero = FALSE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

Encoding T2 / A2 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VCMP{E}<c>.F64 <Dd>, #0.0
VCMP{E}<c>.F32 <Sd>, #0.0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd				1	0	1	sz	E	1	(0)	0	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	1	0	1	Vd				1	0	1	sz	E	1	(0)	0	(0)	(0)	(0)	(0)

```
dp_operation = (sz == '1');  quiet_nan_exc = (E == '1');  with_zero = TRUE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.292 VCMP、VCMPE

この命令は、2 つの浮動小数点レジスタを比較するか、1 つの浮動小数点レジスタと 0 を比較します。結果を FPSCR フラグに書き込みます。これらのフラグは通常、以後の VMRS 命令によって ARM フラグに転送されます。

この命令は、いずれかのオペランドが任意のタイプの NaN である場合に無効演算例外を引き起こすことがあります。いずれかのオペランドがシグナル NaN である場合には、必ず無効演算例外を引き起こします。

エンコード T1/A1 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VCMP{E}<c>.F64 <Dd>, <Dm>
VCMP{E}<c>.F32 <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	0	Vd				1	0	1	sz	E	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	1	0	0	Vd				1	0	1	sz	E	1	M	0	Vm			

```
dp_operation = (sz == '1');  quiet_nan_exc = (E == '1');  with_zero = FALSE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

エンコード T2/A2 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VCMP{E}<c>.F64 <Dd>, #0.0
VCMP{E}<c>.F32 <Sd>, #0.0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd				1	0	1	sz	E	1	(0)	0	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	1	0	1	Vd				1	0	1	sz	E	1	(0)	0	(0)	(0)	(0)	(0)

```
dp_operation = (sz == '1');  quiet_nan_exc = (E == '1');  with_zero = TRUE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCMP{E}<c><q>.F64 <Dd>, <Dm>`	Encoding T1 / A1, sz = 1
`VCMP{E}<c><q>.F32 <Sd>, <Sm>`	Encoding T1 / A1, sz = 0
`VCMP{E}<c><q>.F64 <Dd>, #0.0`	Encoding T2 / A2, sz = 1
`VCMP{E}<c><q>.F32 <Sd>, #0.0`	Encoding T2 / A2, sz = 0

where:

`E` If present, any NaN operand causes an Invalid Operation exception. Encoded as E = 1.
Otherwise, only a signaling NaN causes the exception. Encoded as E = 0.

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Dd>, <Dm>` The operand vectors, for a doubleword operation.

`<Sd>, <Sm>` The operand vectors, for a singleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        op2 = if with_zero then FPZero('0',64) else D[m];
        (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(D[d], op2, quiet_nan_exc, TRUE);
    else
        op2 = if with_zero then FPZero('0',32) else S[m];
        (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(S[d], op2, quiet_nan_exc, TRUE);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Invalid Operation, Input Denormal.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or *unordered*. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This results in the FPSCR flags being set as N=0, Z=0, C=1 and V=1.

`VCMPE` raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCMP{E}<c><q>.F64 <Dd>, <Dm>`	エンコード T1/A1、sz = 1
`VCMP{E}<c><q>.F32 <Sd>, <Sm>`	エンコード T1/A1、sz = 0
`VCMP{E}<c><q>.F64 <Dd>, #0.0`	エンコード T2/A2、sz = 1
`VCMP{E}<c><q>.F32 <Sd>, #0.0`	エンコード T2/A2、sz = 0

各項目の説明については以下を参照して下さい。

`E` 指定した場合、NaN オペランドによって無効演算例外が引き起こされます。E = 1 としてエンコードされます。
省略した場合、シグナル NaN によってのみ無効演算例外が引き起こされます。E = 0 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Dd>, <Dm>` ダブルワード演算の場合のオペランドベクタ

`<Sd>, <Sm>` シングルワード演算の場合のオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        op2 = if with_zero then FPZero('0',64) else D[m];
        (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(D[d], op2, quiet_nan_exc, TRUE);
    else
        op2 = if with_zero then FPZero('0',32) else S[m];
        (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(S[d], op2, quiet_nan_exc, TRUE);
```

例外

未定義命令

浮動小数点例外：無効演算、入力非正規化

NaN

IEEE 754 規格では、比較の結果が <?==、>、または*規則なし*のいずれかでなければならないと規定されています。オペランドの一方または両方が NaN である場合、比較の結果は「規則なし」で、「オペランド 1 < オペランド 2」、「オペランド 1 == オペランド 2」、および「オペランド 1 > オペランド 2」のすべてが FALSE になります。これにより、FPSCR フラグが N=0、Z=0、C=1、V=1 にセットされます。

`VCMPE` は、いずれかのオペランドがどれかのタイプの NaN である場合に無効演算例外を引き起こします。この命令は、<、<=、>、>=、およびオペランドが「規則なし」の場合に例外を引き起こすその他の命題のテストに適しています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.293 VCNT

This instruction counts the number of bits that are one in each element in a vector, and places the results in a second vector.

The operand vector elements must be 8-bit bitfields.

The result vector elements are 8-bit integers.

Encoding T1 / A1 Advanced SIMD

```
VCNT<c>.8 <Qd>, <Qm>
VCNT<c>.8 <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	0	1	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	0	1	0	Q	M	0	Vm			

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8;  elements = 8;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.293 VCNT

VCNT は、ベクタの各エレメントに含まれる 1 のビットの数をカウントし、結果を 2 番目のベクタに配置します。

オペランドベクタのエレメントは、8 ビットのビットフィールドです。

結果ベクタのエレメントは、8 ビットの整数です。

エンコード T1/A1 アドバンスト SIMD

```
VCNT<c>.8 <Qd>, <Qm>
VCNT<c>.8 <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	0	1	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	0	1	0	Q	M	0	Vm			

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8;  elements = 8;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCNT<c><q>.8 <Qd>, <Qm>`	Encoded as Q = 1
`VCNT<c><q>.8 <Dd>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VCNT` instruction must be unconditional.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = BitCount(Elem[D[m+r],e,esize]);
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCNT<c><q>.8 <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VCNT<c><q>.8 <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VCNT` は常に無条件命令です。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = BitCount(Elem[D[m+r],e,esize]);
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.294 VCVT (between floating-point and integer, Advanced SIMD)

This instruction converts each element in a vector from floating-point to integer, or from integer to floating-point, and places the results in a second vector.

The vector elements must be 32-bit floating-point numbers, or 32-bit integers. Signed and unsigned integers are distinct.

The floating-point to integer operation uses the Round towards Zero rounding mode. The integer to floating-point operation uses the Round to Nearest rounding mode.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
VCVT<c>.<Td>.<Tm> <Qd>, <Qm>
VCVT<c>.<Td>.<Tm> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	1	Vd				0	1	1	op		Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	1	Vd				0	1	1	op		Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
to_integer = (op<1> == '1');  unsigned = (op<0> == '1');  esize = 32;  elements = 2;
if to_integer then
    round_zero = TRUE;     // Variable name indicates purpose of FPToFixed() argument
else
    round_nearest = TRUE;  // Variable name indicates purpose of FixedToFP() argument
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.294 VCVT（浮動小数点数と整数との間、アドバンスト SIMD）

この命令は、ベクタの各エレメントを浮動小数点数から整数に、または整数から浮動小数点数に変換し、結果を 2 番目のベクタに配置します。

ベクタのエレメントは、32 ビットの浮動小数点数または 32 ビットの整数の必要があります。整数には、符号付きと符号なしの区別があります。

浮動小数点数から整数への変換演算では、0 への丸めモードが使用されます。整数から浮動小数点数への変換演算では、近似値への丸めモードが使用されます。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VCVT<c>.<Td>.<Tm> <Qd>, <Qm>
VCVT<c>.<Td>.<Tm> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	1	Vd				0	1	1	op		Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	1	Vd				0	1	1	op		Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
to_integer = (op<1> == '1');  unsigned = (op<0> == '1');  esize = 32;  elements = 2;
if to_integer then
    round_zero = TRUE;     // Variable name indicates purpose of FPToFixed() argument
else
    round_nearest = TRUE;  // Variable name indicates purpose of FixedToFP() argument
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VCVT<c><q>.<Td>.<Tm>  <Qd>, <Qm>        Encoded as Q = 1
VCVT<c><q>.<Td>.<Tm>  <Dd>, <Dm>        Encoded as Q = 0
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VCVT` instruction must be unconditional.

`.<Td>.<Tm>` The data types for the elements of the vectors. They must be one of:

- `.S32.F32` encoded as op = 0b10, size = 0b10
- `.U32.F32` encoded as op = 0b11, size = 0b10
- `.F32.S32` encoded as op = 0b00, size = 0b10
- `.F32.U32` encoded as op = 0b01, size = 0b10.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op = Elem[D[m+r],e,esize];
            if to_integer then
                result = FPToFixed(op, esize, 0, unsigned, round_zero, FALSE);
            else
                result = FixedToFP(op, esize, 0, unsigned, round_nearest, FALSE);
            Elem[D[d+r],e,esize] = result;
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VCVT<c><q>.<Td>.<Tm>  <Qd>, <Qm>        Q = 1 としてエンコードされる
VCVT<c><q>.<Td>.<Tm>  <Dd>, <Dm>        Q = 0 としてエンコードされる
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VCVT` は常に無条件命令です。

`.<Td>.<Tm>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `.S32.F32` op = 0b10、size = 0b10 としてエンコードされます。
- `.U32.F32` op = 0b11、size = 0b10 としてエンコードされます。
- `.F32.S32` op = 0b00、size = 0b10 としてエンコードされます。
- `.F32.U32` op = 0b01、size = 0b10 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op = Elem[D[m+r],e,esize];
            if to_integer then
                result = FPToFixed(op, esize, 0, unsigned, round_zero, FALSE);
            else
                result = FixedToFP(op, esize, 0, unsigned, round_nearest, FALSE);
            Elem[D[d+r],e,esize] = result;
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.295 VCVT, VCVTR (between floating-point and integer, VFP)

These instructions convert a value in a register from floating-point to a 32-bit integer, or from a 32-bit integer to floating-point, and place the result in a second register.

The floating-point to integer operation normally uses the Round towards Zero rounding mode, but can optionally use the rounding mode specified by the FPSCR. The integer to floating-point operation uses the rounding mode specified by the FPSCR.

VCVT (between floating-point and fixed-point, VFP) on page A8-582 describes conversions between floating-point and 16-bit integers.

Encoding T1 / A1 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VCVT{R}<c>.S32.F64 <Sd>, <Dm>
VCVT{R}<c>.S32.F32 <Sd>, <Sm>
VCVT{R}<c>.U32.F64 <Sd>, <Dm>
VCVT{R}<c>.U32.F32 <Sd>, <Sm>
VCVT<c>.F64.<Tm> <Dd>, <Sm>
VCVT<c>.F32.<Tm> <Sd>, <Sm>
```

15 14 13 12	11 10 9 8	7	6	5 4	3	2 1 0	15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0
1 1 1 0	1 1 1 0	1	D	1 1	1	opc2	Vd	1 0 1	sz	op	1	M	0	Vm

31 30 29 28	27 26 25 24	23	22	21 20	19	18 17 16	15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0
cond	1 1 1 0	1	D	1 1	1	opc2	Vd	1 0 1	sz	op	1	M	0	Vm

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
to_integer = (opc2<2> == '1');  dp_operation = (sz == 1);
if to_integer then
    unsigned = (opc2<0> == '0');  round_zero = (op == '1');
    d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
else
    unsigned = (op == '0');  round_fpscr = FALSE;  // FALSE selects FPSCR rounding
    m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```

Related encodings See *VFP data-processing instructions* on page A7-24

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.295 VCVT、VCVTR（浮動小数点数と整数との間、VFP）

VCVT、VCVTR（浮動小数点数と整数との間、VFP）は、レジスタの値を浮動小数点数から 32 ビット整数に、または 32 ビット整数から浮動小数点数に変換し、結果を 2 番目のレジスタに配置します。

浮動小数点数から整数への変換演算では通常、0 への丸めモードが使用されますが、オプションとして FPSCR で指定された丸めモードを使用することもできます。整数から浮動小数点数への変換演算では、FPSCR で指定された丸めモードが使用されます。

浮動小数点数と 16 ビット整数との間の変換については、*VCVT（浮動小数点数と固定小数点数との間、VFP）* P. A8- 582 を参照して下さい。

エンコード T1/A1 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VCVT{R}<c>.S32.F64 <Sd>, <Dm>
VCVT{R}<c>.S32.F32 <Sd>, <Sm>
VCVT{R}<c>.U32.F64 <Sd>, <Dm>
VCVT{R}<c>.U32.F32 <Sd>, <Sm>
VCVT<c>.F64.<Tm> <Dd>, <Sm>
VCVT<c>.F32.<Tm> <Sd>, <Sm>
```

15 14 13 12	11 10 9 8	7	6	5 4	3	2 1 0	15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0
1 1 1 0	1 1 1 0	1	D	1 1	1	opc2	Vd	1 0 1	sz	op	1	M	0	Vm

31 30 29 28	27 26 25 24	23	22	21 20	19	18 17 16	15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0
cond	1 1 1 0	1	D	1 1	1	opc2	Vd	1 0 1	sz	op	1	M	0	Vm

```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
to_integer = (opc2<2> == '1');  dp_operation = (sz == 1);
if to_integer then
    unsigned = (opc2<0> == '0');  round_zero = (op == '1');
    d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
else
    unsigned = (op == '0');  round_fpscr = FALSE;  // FALSE selects FPSCR rounding
    m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
```

関連エンコード *VFP のデータ処理命令* P. A7- 24 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCVT{R}<c><q>.S32.F64 <Sd>, <Dm>`	opc2 = '101', sz = 1
`VCVT{R}<c><q>.S32.F32 <Sd>, <Sm>`	opc2 = '101', sz = 0
`VCVT{R}<c><q>.U32.F64 <Sd>, <Dm>`	opc2 = '100', sz = 1
`VCVT{R}<c><q>.U32.F32 <Sd>, <Sm>`	opc2 = '100', sz = 0
`VCVT<c><q>.F64.<Tm> <Dd>, <Sm>`	opc2 = '000', sz = 1
`VCVT<c><q>.F32.<Tm> <Sd>, <Sm>`	opc2 = '000', sz = 0

where:

`R` If `R` is specified, the operation uses the rounding mode specified by the FPSCR. Encoded as op = 0.

If `R` is omitted. the operation uses the Round towards Zero rounding mode. For syntaxes in which `R` is optional, op is encoded as 1 if `R` is omitted.

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Tm>` The data type for the operand. It must be one of:

- `S32` encoded as op = 1
- `U32` encoded as op = 0.

`<Sd>, <Dm>` The destination register and the operand register, for a double-precision operand.

`<Dd>, <Sm>` The destination register and the operand register, for a double-precision result.

`<Sd>, <Sm>` The destination register and the operand register, for a single-precision operand or result.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        if dp_operation then
            S[d] = FPToFixed(D[m], 32, 0, unsigned, round_zero, TRUE);
        else
            S[d] = FPToFixed(S[m], 32, 0, unsigned, round_zero, TRUE);
    else
        if dp_operation then
            D[d] = FixedToFP(S[m], 64, 0, unsigned, round_fpscr, TRUE);
        else
            S[d] = FixedToFP(S[m], 32, 0, unsigned, round_fpscr, TRUE);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCVT{R}<c><q>.S32.F64 <Sd>, <Dm>`	opc2 = '101', sz = 1
`VCVT{R}<c><q>.S32.F32 <Sd>, <Sm>`	opc2 = '101', sz = 0
`VCVT{R}<c><q>.U32.F64 <Sd>, <Dm>`	opc2 = '100', sz = 1
`VCVT{R}<c><q>.U32.F32 <Sd>, <Sm>`	opc2 = '100', sz = 0
`VCVT<c><q>.F64.<Tm> <Dd>, <Sm>`	opc2 = '000', sz = 1
`VCVT<c><q>.F32.<Tm> <Sd>, <Sm>`	opc2 = '000', sz = 0

各項目の説明については以下を参照して下さい。

`R` `R` を指定した場合、FPSCR で指定された丸めモードが使用されます。op = 0 としてエンコードされます。

`R` を省略した場合、0 への丸めモードが使用されます。`R` がオプションである構文では、`R` を省略した場合、op は 1 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Tm>` オペランドのデータ型。次のいずれかを使用できます。

- `S32` op = 1 としてエンコードされます。
- `U32` op = 0 としてエンコードされます。

`<Sd>, <Dm>` オペランドが倍精度の場合のデスティネーションレジスタとオペランドレジスタ

`<Dd>, <Sm>` 結果が倍精度の場合のデスティネーションレジスタとオペランドレジスタ

`<Sd>, <Sm>` オペランドまたは結果が単精度の場合のデスティネーションレジスタとオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        if dp_operation then
            S[d] = FPToFixed(D[m], 32, 0, unsigned, round_zero, TRUE);
        else
            S[d] = FPToFixed(S[m], 32, 0, unsigned, round_zero, TRUE);
    else
        if dp_operation then
            D[d] = FixedToFP(S[m], 64, 0, unsigned, round_fpscr, TRUE);
        else
            S[d] = FixedToFP(S[m], 32, 0, unsigned, round_fpscr, TRUE);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.296 VCVT (between floating-point and fixed-point, Advanced SIMD)

This instruction converts each element in a vector from floating-point to fixed-point, or from fixed-point to floating-point, and places the results in a second vector.

The vector elements must be 32-bit floating-point numbers, or 32-bit integers. Signed and unsigned integers are distinct.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
VCVT<c>.<Td>.<Tm> <Qd>, <Qm>, #<fbits>
VCVT<c>.<Td>.<Tm> <Dd>, <Dm>, #<fbits>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				1	1	1	op	0	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				1	1	1	op	0	Q	M	1	Vm			

```
if imm6 == '000xxx' then SEE "Related encodings";
if imm6 == '0xxxxx' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op == '1');  unsigned = (U == '1');
if to_fixed then
    round_zero = TRUE;     // Variable name indicates purpose of FPToFixed() argument
else
    round_nearest = TRUE;  // Variable name indicates purpose of FixedToFP() argument
esize = 32;  frac_bits = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings See *One register and a modified immediate value* on page A7-21

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.296 VCVT（浮動小数点数と固定小数点数との間、アドバンスト SIMD）

VCVT（浮動小数点数と固定小数点数との間、アドバンスト SIMD）は、ベクタの各エレメントを浮動小数点数から固定小数点数に、または固定小数点数から浮動小数点数に変換し、結果を 2 番目のベクタに配置します。

ベクタのエレメントは、32 ビットの浮動小数点数または 32 ビットの整数の必要があります。整数には、符号付きと符号なしの区別があります。

浮動小数点数から固定小数点数への変換演算では、0 への丸めモードが使用されます。固定小数点数から浮動小数点数への変換演算では、近似値への丸めモードが使用されます。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VCVT<c>.<Td>.<Tm> <Qd>, <Qm>, #<fbits>
VCVT<c>.<Td>.<Tm> <Dd>, <Dm>, #<fbits>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				1	1	1	op	0	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				1	1	1	op	0	Q	M	1	Vm			

```
if imm6 == '000xxx' then SEE "Related encodings";
if imm6 == '0xxxxx' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op == '1');  unsigned = (U == '1');
if to_fixed then
    round_zero = TRUE;     // Variable name indicates purpose of FPToFixed() argument
else
    round_nearest = TRUE;  // Variable name indicates purpose of FixedToFP() argument
esize = 32;  frac_bits = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCVT<c><q>.<Td>.<Tm>  <Qd>, <Qm>, #<fbits>`	Encoded as Q = 1
`VCVT<c><q>.<Td>.<Tm>  <Dd>, <Dm>, #<fbits>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VCVT` instruction must be unconditional.

`.<Td>.<Tm>` The data types for the elements of the vectors. They must be one of:

- `.S32.F32` encoded as op = 1, U = 0
- `.U32.F32` encoded as op = 1, U = 1
- `.F32.S32` encoded as op = 0, U = 0
- `.F32.U32` encoded as op = 0, U = 1.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

`<fbits>` The number of fraction bits in the fixed point number, in the range 1 to 32:

- (64 - `<fbits>`) is encoded in imm6.

An assembler can permit an `<fbits>` value of 0. This is encoded as floating-point to integer or integer to floating-point instruction, see *VCVT (between floating-point and integer, Advanced SIMD)* on page A8-576.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op = Elem[D[m+r],e,esize];
            if to_fixed then
                result = FPToFixed(op, esize, frac_bits, unsigned, round_zero, FALSE);
            else
                result = FixedToFP(op, esize, frac_bits, unsigned, round_nearest, FALSE);
            Elem[D[d+r],e,esize] = result;
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCVT<c><q>.<Td>.<Tm>  <Qd>, <Qm>, #<fbits>`	Q = 1 としてエンコードされる
`VCVT<c><q>.<Td>.<Tm>  <Dd>, <Dm>, #<fbits>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VCVT` は常に無条件命令です。

`.<Td>.<Tm>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `.S32.F32` op = 1、U = 0 としてエンコードされます。
- `.U32.F32` op = 1、U = 1 としてエンコードされます。
- `.F32.S32` op = 0、U = 0 としてエンコードされます。
- `.F32.U32` op = 0、U = 1 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<fbits>` 固定小数点数の仮数ビットの数で、範囲は 1 ～ 32 です。

- (64 − `<fbits>`) が imm6 にエンコードされます。

アセンブラは、`<fbits>` の値として 0 を許可できます。これは、浮動小数点数から整数への変換命令、または整数から浮動小数点数への変換命令としてエンコードされます。詳細については、*VCVT (浮動小数点数と整数との間、アドバンスト SIMD)* P. A8-576 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op = Elem[D[m+r],e,esize];
            if to_fixed then
                result = FPToFixed(op, esize, frac_bits, unsigned, round_zero, FALSE);
            else
                result = FixedToFP(op, esize, frac_bits, unsigned, round_nearest, FALSE);
            Elem[D[d+r],e,esize] = result;
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.297 VCVT (between floating-point and fixed-point, VFP)

This instruction converts a value in a register from floating-point to fixed-point, or from fixed-point to floating-point, and places the result in a second register. You can specify the fixed-point value as either signed or unsigned.

The floating-point value can be single-precision or double-precision.

The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits. Signed conversions to fixed-point values sign-extend the result value to the destination register width. Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Encoding T1 / A1 VFPv3 (sf = 1 UNDEFINED in single-precision only variants)

```
VCVT<c>.<Td>.F64 <Dd>, <Dd>, #<fbits>
VCVT<c>.<Td>.F32 <Sd>, <Sd>, #<fbits>
VCVT<c>.F64.<Td> <Dd>, <Dd>, #<fbits>
VCVT<c>.F32.<Td> <Sd>, <Sd>, #<fbits>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	1	op	1	U	Vd				1	0	1	sf	sx	1	i	0	imm4			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	1	op	1	U	Vd				1	0	1	sf	sx	1	i	0	imm4			

```
to_fixed = (op == '1');  dp_operation = (sf == '1');  unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
if to_fixed then
    round_zero = TRUE;
else
    round_nearest = TRUE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
if frac_bits < 0 then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.297 VCVT（浮動小数点数と固定小数点数との間、VFP）

VCVT（浮動小数点数と固定小数点数との間、VFP）は、レジスタの値を浮動小数点数から固定小数点数に、または固定小数点数から浮動小数点数に変換し、結果を 2 番目のレジスタに配置します。固定小数点数として、符号付きまたは符号なしの値を指定できます。

浮動小数点数としては、単精度または倍精度の値を指定できます。

固定小数点数は 16 ビットまたは 32 ビットです。固定小数点数からの変換では、ソースレジスタの下位ビットからオペランドが取得され、残りのビットはすべて無視されます。固定小数点数への符号付き変換では、結果の値がデスティネーションレジスタの幅に符号拡張されます。固定小数点数への符号なし変換では、結果の値がデスティネーションレジスタの幅にゼロ拡張されます。

浮動小数点数から固定小数点数への変換演算では、0 への丸めモードが使用されます。固定小数点数から浮動小数点数への変換演算では、近似値への丸めモードが使用されます。

エンコード T1/A1 VFPv3（単精度のみのバリアントでは sf = 1 は未定義）

```
VCVT<c>.<Td>.F64 <Dd>, <Dd>, #<fbits>
VCVT<c>.<Td>.F32 <Sd>, <Sd>, #<fbits>
VCVT<c>.F64.<Td> <Dd>, <Dd>, #<fbits>
VCVT<c>.F32.<Td> <Sd>, <Sd>, #<fbits>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	1	op	1	U	Vd				1	0	1	sf	sx	1	i	0	imm4			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	1	op	1	U	Vd				1	0	1	sf	sx	1	i	0	imm4			

```
to_fixed = (op == '1');  dp_operation = (sf == '1');  unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
if to_fixed then
    round_zero = TRUE;
else
    round_nearest = TRUE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
if frac_bits < 0 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VCVT<c><q>.<Td>.F64  <Dd>, <Dd>,#<fbits>          op = 1, sf = 1
VCVT<c><q>.<Td>.F32  <Sd>, <Sd>, #<fbits>         op = 1, sf = 0
VCVT<c><q>.F64.<Td>  <Dd>, <Dd>, #<fbits>         op = 0, sf = 1
VCVT<c><q>.F32.<Td>  <Sd>, <Sd>, #<fbits>         op = 0, sf = 0
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Td>` The data type for the fixed-point number. It must be one of:

- `S16` encoded as U = 0, sx = 0
- `U16` encoded as U = 1, sx = 0
- `S32` encoded as U = 0, sx = 1
- `U32` encoded as U = 1, sx = 1.

`<Dd>` The destination and operand register, for a double-precision operand.

`<Sd>` The destination and operand register, for a single-precision operand.

`<fbits>` The number of fraction bits in the fixed-point number:

- If `<Td>` is `S16` or `U16`, `<fbits>` must be in the range 0-16. (16 - `<fbits>`) is encoded in [imm4,i]
- I f `<Td>` is `S32` or `U32`, `<fbits>` must be in the range 1-32. (32 - `<fbits>`) is encoded in [imm4,i].

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_fixed then
        if dp_operation then
            result = FPToFixed(D[d], size, frac_bits, unsigned, round_zero, TRUE);
            D[d] = if unsigned then ZeroExtend(result, 64) else SignExtend(result, 64);
        else
            result = FPToFixed(S[m], size, frac_bits, unsigned, round_zero, TRUE);
            S[d] = if unsigned then ZeroExtend(result, 32) else SignExtend(result, 32);
    else
        if dp_operation then
            D[d] = FixedToFP(D[d]<size-1:0>, 64, frac_bits, unsigned, round_nearest, TRUE);
        else
            S[d] = FixedToFP(S[d]<size-1:0>, 32, frac_bits, unsigned, round_nearest, TRUE);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VCVT<c><q>.<Td>.F64  <Dd>, <Dd>,#<fbits>          op = 1、sf = 1
VCVT<c><q>.<Td>.F32  <Sd>, <Sd>, #<fbits>         op = 1、sf = 0
VCVT<c><q>.F64.<Td>  <Dd>, <Dd>, #<fbits>         op = 0、sf = 1
VCVT<c><q>.F32.<Td>  <Sd>, <Sd>, #<fbits>         op = 0、sf = 0
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Td>` 固定小数点数のデータ型。次のいずれかを使用できます。

- `S16` U = 0、sx = 0 としてエンコードされます。
- `U16` U = 1、sx = 0 としてエンコードされます。
- `S32` U = 0、sx = 1 としてエンコードされます。
- `U32` U = 1、sx = 1 としてエンコードされます。

`<Dd>` オペランドが倍精度の場合のデスティネーションレジスタとオペランドレジスタ

`<Sd>` オペランドが単精度の場合のデスティネーションレジスタとオペランドレジスタ

`<fbits>` 固定小数点数の仮数ビットの数。

- `<Td>` が `S16` または `U16` の場合、`<fbits>` の範囲は 0 ～ 16 です。(16 − `<fbits>`) が [imm4,i] にエンコードされます。
- `<Td>` が `S32` または `U32` の場合、`<fbits>` の範囲は 1 ～ 32 です。(32 − `<fbits>`) が [imm4,i] にエンコードされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_fixed then
        if dp_operation then
            result = FPToFixed(D[d], size, frac_bits, unsigned, round_zero, TRUE);
            D[d] = if unsigned then ZeroExtend(result, 64) else SignExtend(result, 64);
        else
            result = FPToFixed(S[m], size, frac_bits, unsigned, round_zero, TRUE);
            S[d] = if unsigned then ZeroExtend(result, 32) else SignExtend(result, 32);
    else
        if dp_operation then
            D[d] = FixedToFP(D[d]<size-1:0>, 64, frac_bits, unsigned, round_nearest, TRUE);
        else
            S[d] = FixedToFP(S[d]<size-1:0>, 32, frac_bits, unsigned, round_nearest, TRUE);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.298 VCVT (between double-precision and single-precision)

This instruction does one of the following:

- converts the value in a double-precision register to single-precision and writes the result to a single-precision register
- converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

Encoding T1 / A1 VFPv2, VFPv3 (UNDEFINED in single-precision only variants)

```
VCVT<c>.F64.F32 <Dd>, <Sm>
VCVT<c>.F32.F64 <Sd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	1	1	Vd				1	0	1	sz	1	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	1	1	1	Vd				1	0	1	sz	1	1	M	0	Vm			

```
double_to_single = (sz == '1');
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.298 VCVT（倍精度と単精度との間）

VCVT（倍精度と単精度との間）は、次のいずれかを実行します。

- 倍精度レジスタの値を単精度に変換し、結果を単精度レジスタに書き込む。
- 単精度レジスタの値を倍精度に変換し、結果を倍精度レジスタに書き込む。

エンコード T1/A1 VFPv2、VFPv3（単精度のみのバリアントでは未定義）

```
VCVT<c>.F64.F32 <Dd>, <Sm>
VCVT<c>.F32.F64 <Sd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	1	1	Vd				1	0	1	sz	1	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	1	1	1	Vd				1	0	1	sz	1	1	M	0	Vm			

```
double_to_single = (sz == '1');
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCVT<c><q>.F64.F32 <Dd>, <Sm>`	Encoded as sz = 0
`VCVT<c><q>.F32.F64 <Sd>, <Dm>`	Encoded as sz = 1

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Dd>, <Sm>` The destination register and the operand register, for a single-precision operand.

`<Sd>, <Dm>` The destination register and the operand register, for a double-precision operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if double_to_single then
        S[d] = FPDoubleToSingle(D[m], TRUE);
    else
        D[d] = FPSingleToDouble(S[m], TRUE);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Invalid Operation, Input Denormal, Overflow, Underflow, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`VCVT<c><q>.F64.F32 <Dd>, <Sm>`	sz = 0 としてエンコードされる
`VCVT<c><q>.F32.F64 <Sd>, <Dm>`	sz = 1 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Dd>, <Sm>` オペランドが単精度の場合のデスティネーションレジスタとオペランドレジスタ

`<Sd>, <Dm>` オペランドが倍精度の場合のデスティネーションレジスタとオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if double_to_single then
        S[d] = FPDoubleToSingle(D[m], TRUE);
    else
        D[d] = FPSingleToDouble(S[m], TRUE);
```

例外

未定義命令

浮動小数点例外：無効演算、入力非正規化、オーバフロー、アンダーフロー、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.299 VCVT (between half-precision and single-precision, Advanced SIMD)

This instruction converts each element in a vector from single-precision to half-precision floating-point or from half-precision to single-precision, and places the results in a second vector.

The vector elements must be 32-bit floating-point numbers, or 16-bit floating-point numbers.

Encoding T1 / A1 Advanced SIMD with half-precision extensions (UNDEFINED in integer-only variant)

```
VCVT<c>.F32.F16 <Qd>, <Dm>
VCVT<c>.F16.F32 <Dd>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	1	1	op	0	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	1	1	op	0	0	M	0	Vm			

```
half_to_single = (op == '1');
if size != '01' then UNDEFINED;
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
esize = 16;  elements = 4;
m = UInt(M:Vm);  d = UInt(D:Vd);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.299 VCVT（半精度と単精度との間、アドバンスト SIMD）

CVT（半精度と単精度との間、アドバンスト SIMD）は、ベクタの各エレメントを単精度浮動小数点数から半精度浮動小数点数に、または半精度浮動小数点数から単精度浮動小数点数に変換し、結果を 2 番目のベクタに配置します。

ベクタのエレメントは、32 ビットの浮動小数点数または 16 ビットの浮動小数点の必要があります。

エンコード T1/A1 半精度拡張機能を含むアドバンスト SIMD（整数のみのバリアントでは未定義）

```
VCVT<c>.F32.F16 <Qd>, <Dm>
VCVT<c>.F16.F32 <Dd>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	1	1	op	0	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	1	1	op	0	0	M	0	Vm			

```
half_to_single = (op == '1');
if size != '01' then UNDEFINED;
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
esize = 16;  elements = 4;
m = UInt(M:Vm);  d = UInt(D:Vd);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCVT<c><q>.F32.F16 <Qd>, <Dm>`	Encoded as op = 1
`VCVT<c><q>.F16.F32 <Dd>, <Qm>`	Encoded as op = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Qd>, <Dm>` The destination vector and the operand vector for a half-precision to single-precision operation.

`<Dd>, <Qm>` The destination vector and the operand vectors for a single-precision to half-precision operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if half_to_single then
            Elem[Q[d>>1],e,2*esize] = FPHalfToSingle(Elem[D[m],e,esize], FALSE);
        else
            Elem[D[d],e,esize] = FPSingleToHalf(Elem[Q[m>>1],e,2*esize], FALSE);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Invalid Operation, Input Denormal, Overflow, Underflow, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCVT<c><q>.F32.F16 <Qd>, <Dm>`	op = 1 としてエンコードされる
`VCVT<c><q>.F16.F32 <Dd>, <Qm>`	op = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Qd>, <Dm>` 半精度から単精度への変換演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Qm>` 単精度から半精度への変換演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if half_to_single then
            Elem[Q[d>>1],e,2*esize] = FPHalfToSingle(Elem[D[m],e,esize], FALSE);
        else
            Elem[D[d],e,esize] = FPSingleToHalf(Elem[Q[m>>1],e,2*esize], FALSE);
```

例外

未定義命令

浮動小数点例外：無効演算、入力非正規化、オーバフロー、アンダーフロー、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.300 VCVTB, VCVTT (between half-precision and single-precision, VFP)

This instruction does one of the following:

- converts the half-precision value in the top or bottom half of a single-precision register to single-precision and writes the result to a single-precision register
- converts the value in a single-precision register to half-precision and writes the result into the top or bottom half of a single-precision register, preserving the other half of the target register.

Encoding T1 / A1 VFPv3 half-precision extensions

```
VCVT<y><c>.F32.F16 <Sd>, <Sm>
VCVT<y><c>.F16.F32 <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	op	Vd				1	0	1	0	T	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	0	1	op	Vd				1	0	1	0	T	1	M	0	Vm			

```
half_to_single = (op == '0');
lowbit = if T == '1' then 16 else 0;
m = UInt(Vm:M);  d = UInt(Vd:D);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.300 VCVTB、VCVTT（半精度と単精度との間、VFP）

VCVTB、VCVTT（半精度と単精度との間、VFP）は、次のいずれかを実行します。

- 単精度レジスタの上位半分または下位半分の半精度値を単精度に変換し、結果を単精度レジスタに書き込む。
- 単精度レジスタの値を半精度に変換し、結果を単精度レジスタの上位半分または下位半分に書き込む。ターゲットレジスタの他方の半分は変更されません。

エンコード T1/A1 VFPv3 の半精度拡張機能

```
VCVT<y><c>.F32.F16 <Sd>, <Sm>
VCVT<y><c>.F16.F32 <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	op	Vd				1	0	1	0	T	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	0	1	op	Vd				1	0	1	0	T	1	M	0	Vm			

```
half_to_single = (op == '0');
lowbit = if T == '1' then 16 else 0;
m = UInt(Vm:M);  d = UInt(Vd:D);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VCVT<y><c><q>.F32.F16 <Sd>, <Sm>`	Encoded as op = 0
`VCVT<y><c><q>.F16.F32 <Sd>, <Sm>`	Encoded as op = 1

where:

`<y>` Specifies which half of the operand register `<Sm>` or destination register `<Sd>` is used for the operand or destination. If `<y>` is B, then the T bit is encoded as 0 and the bottom half (bits [15:0]) of `<Sm>` or `<Sd>` is used. If `<y>` is T, then the T bit is encoded as 1 and the top half (bits [31:16]) of `<Sm>` or `<Sd>` is used

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Sd>` The destination register.

`<Sm>` The operand register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if half_to_single then
        S[d] = FPHalfToSingle(S[m]<lowbit+15:lowbit>, TRUE);
    else
        S[d]<lowbit+15:lowbit> = FPSingleToHalf(S[m], TRUE);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Invalid Operation, Input Denormal, Overflow, Underflow, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VCVT<y><c><q>.F32.F16 <Sd>, <Sm>`	op = 0 としてエンコードされる
`VCVT<y><c><q>.F16.F32 <Sd>, <Sm>`	op = 1 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<y>` オペランドまたはデスティネーションとして、オペランドレジスタ `<Sm>` またはデスティネーションレジスタ `<Sd>` のどの半分を使用するかを指定します。`<y>` が B の場合、T ビットが 0 としてエンコードされ、`<Sm>` または `<Sd>` の下位半分（ビット [15:0]）が使用されます。`<y>` が T の場合、T ビットが 1 としてエンコードされ、`<Sm>` または `<Sd>` の上位半分（ビット [31:16]）が使用されます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Sd>` デスティネーションレジスタ

`<Sm>` オペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if half_to_single then
        S[d] = FPHalfToSingle(S[m]<lowbit+15:lowbit>, TRUE);
    else
        S[d]<lowbit+15:lowbit> = FPSingleToHalf(S[m], TRUE);
```

例外

未定義命令

浮動小数点例外：無効演算、入力非正規化、オーバフロー、アンダーフロー、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.301 VDIV

This instruction divides one floating-point value by another floating-point value and writes the result to a third floating-point register.

Encoding T1 / A1 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VDIV<c>.F64 <Dd>, <Dn>, <Dm>
VDIV<c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	0	0	Vn				Vd				1	0	1	sz	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	0	0	Vn				Vd				1	0	1	sz	N	0	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP vectors This instruction can operate on VFP vectors under control of the FPSCR.LEN and FPSCR.STRIDE bits. For details see Appendix F *VFP Vector Operation Support*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.301 VDIV

VDIV（浮動小数点除算）は、浮動小数点数を別の浮動小数点数で除算し、結果を 3 番目の浮動小数点レジスタに書き込みます。

エンコード T1/A1 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VDIV<c>.F64 <Dd>, <Dn>, <Dm>
VDIV<c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	0	0	Vn				Vd				1	0	1	sz	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	0	0	Vn				Vd				1	0	1	sz	N	0	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP ベクタ この命令は、FPSCR.LEN ビットと FPSCR.STRIDE ビットの制御の下で VFP に作用することができます。詳細については、付録 F *「VFP ベクタ演算のサポート」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VDIV<c><q>.F64 {<Dd>,} <Dn>, <Dm>`	Encoded as sz = 1
`VDIV<c><q>.F32 {<Sd>,} <Sn>, <Sm>`	Encoded as sz = 0

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<Dd>, <Dn>, <Dm>`	The destination register and the operand registers, for a double-precision operation.
`<Sd>, <Sn>, <Sm>`	The destination register and the operand registers, for a single-precision operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        D[d] = FPDiv(D[n], D[m], TRUE);
    else
        S[d] = FPDiv(S[n], S[m], TRUE);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Invalid Operation, Division by Zero, Overflow, Underflow, Inexact, Input Denormal.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VDIV<c><q>.F64 {<Dd>,} <Dn>, <Dm>`	sz = 1 としてエンコードされる
`VDIV<c><q>.F32 {<Sd>,} <Sn>, <Sm>`	sz = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<Dd>, <Dn>, <Dm>`	倍精度演算の場合のデスティネーションレジスタとオペランドレジスタ
`<Sd>, <Sn>, <Sm>`	単精度演算の場合のデスティネーションレジスタとオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        D[d] = FPDiv(D[n], D[m], TRUE);
    else
        S[d] = FPDiv(S[n], S[m], TRUE);
```

Exceptions

未定義命令

浮動小数点例外：無効演算、0 による除算、オーバフロー、アンダーフロー、不正確、入力非正規化

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.302 VDUP (scalar)

Vector Duplicate duplicates a scalar into every element of the destination vector.

The scalar, and the destination vector elements, can be any one of 8-bit, 16-bit, or 32-bit bitfields. There is no distinction between data types.

For more information about scalars see *Advanced SIMD scalars* on page A7-9.

Encoding T1 / A1 Advanced SIMD

```
VDUP<c>.<size>  <Qd>, <Dm[x]>
VDUP<c>.<size>  <Dd>, <Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	imm4				Vd				1	1	0	0	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	imm4				Vd				1	1	0	0	0	Q	M	0	Vm			

```
if imm4 == ‘x000’ then UNDEFINED;
if Q == ‘1’ && Vd<0> == ‘1’ then UNDEFINED;
case imm4 of
    when ‘xxx1’  esize = 8;  elements = 8;  index = UInt(imm4<3:1>);
    when ‘xx10’  esize = 16;  elements = 4;  index = UInt(imm4<3:2>);
    when ‘x100’  esize = 32;  elements = 2;  index = UInt(imm4<3>);
d = UInt(D:Vd);  regs = if U == ‘0’ then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.302 VDUP（スカラ）

VDUP（ベクタ複製（スカラ））は、スカラをデスティネーションベクタのすべてのエレメントに複製します。

スカラとデスティネーションベクタのエレメントには、8 ビット、16 ビット、または 32 ビットのビットフィールドを使用できます。これらにはデータ型の区別はありません。

スカラの詳細については、*アドバンスト SIMD スカラ* P. A7-9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VDUP<c>.<size>  <Qd>, <Dm[x]>
VDUP<c>.<size>  <Dd>, <Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	imm4				Vd				1	1	0	0	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	imm4				Vd				1	1	0	0	0	Q	M	0	Vm			

```
if imm4 == ‘x000’ then UNDEFINED;
if Q == ‘1’ && Vd<0> == ‘1’ then UNDEFINED;
case imm4 of
    when ‘xxx1’  esize = 8;  elements = 8;  index = UInt(imm4<3:1>);
    when ‘xx10’  esize = 16;  elements = 4;  index = UInt(imm4<3:2>);
    when ‘x100’  esize = 32;  elements = 2;  index = UInt(imm4<3>);
d = UInt(D:Vd);  regs = if U == ‘0’ then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VDUP<c><q>.<size>  <Qd>, <Dm[x]>`	Encoded as Q = 1
`VDUP<c><q>.<size>  <Dd>, <Dm[x]>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VDUP` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` Encoded as imm4<0> = '1'. imm4<3:1> encodes the index `[x]` of the scalar.
- `16` Encoded as imm4<1:0> = '10'. imm4<3:2> encodes the index `[x]` of the scalar.
- `32` Encoded as imm4<2:0> = '100'. imm4<3> encodes the index `[x]` of the scalar.

`<Qd>` The destination vector for a quadword operation.

`<Dd>` The destination vector for a doubleword operation.

`<Dm[x]>` The scalar. For details of how `[x]` is encoded, see the description of `<size>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    scalar = Elem[D[m],index,esize];
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VDUP<c><q>.<size>  <Qd>, <Dm[x]>`	Q = 1 としてエンコードされる
`VDUP<c><q>.<size>  <Dd>, <Dm[x]>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VDUP` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- `8` imm4〈0〉 = '1' としてエンコードされます。imm4〈3:1〉では、スカラのインデクス `[x]` がエンコードされます。
- `16` imm4〈1:0〉 = '10' としてエンコードされます。imm4〈3:2〉では、スカラのインデクス `[x]` がエンコードされます。
- `32` imm4〈2:0〉 = '100' としてエンコードされます。imm4〈3〉では、スカラのインデクス `[x]` がエンコードされます。

`<Qd>` クワッドワード演算の場合のデスティネーションベクタ

`<Dd>` ダブルワード演算の場合のデスティネーションベクタ

`<Dm[x]>` スカラ。`[x]` のエンコード方法の詳細については、`<size>` の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    scalar = Elem[D[m],index,esize];
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.303 VDUP (ARM core register)

This instruction duplicates an element from an ARM core register into every element of the destination vector.

The destination vector elements can be 8-bit, 16-bit, or 32-bit bitfields. The source element is the least significant 8, 16, or 32 bits of the ARM core register. There is no distinction between data types.

Encoding T1 / A1 Advanced SIMD

```
VDUP<c>.<size>  <Qd>, <Rt>
VDUP<c>.<size>  <Dd>, <Rt>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	b	Q	0	Vd				Rt				1	0	1	1	D	0	e	1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	b	Q	0	Vd				Rt				1	0	1	1	D	0	e	1	(0)	(0)	(0)	(0)

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);  regs = if Q == '0' then 1 else 2;
case b:e of
    when '00'  esize = 32;  elements = 2;
    when '01'  esize = 16;  elements = 4;
    when '10'  esize = 8;  elements = 8;
    when '11'  UNDEFINED;
if t == 15 || (CurrentInstrSet() != InstrSet_ARM && t == 13) then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.303 VDUP（ARM コアレジスタ）

VDUP（ARM コアレジスタ）は、ARM コアレジスタのエレメントをデスティネーションベクタのすべてのエレメントに複製します。

デスティネーションベクタのエレメントには、8 ビット、16 ビット、または 32 ビットのビットフィールドを使用できます。ソースのエレメントは、ARM コアレジスタの最下位 8 ビット、16 ビット、または 32 ビットです。これらにはデータ型の区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VDUP<c>.<size>  <Qd>, <Rt>
VDUP<c>.<size>  <Dd>, <Rt>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	b	Q	0	Vd				Rt				1	0	1	1	D	0	e	1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	b	Q	0	Vd				Rt				1	0	1	1	D	0	e	1	(0)	(0)	(0)	(0)

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);  regs = if Q == '0' then 1 else 2;
case b:e of
    when '00'  esize = 32;  elements = 2;
    when '01'  esize = 16;  elements = 4;
    when '10'  esize = 8;  elements = 8;
    when '11'  UNDEFINED;
if t == 15 || (CurrentInstrSet() != InstrSet_ARM && t == 13) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VDUP<c><q>.<size>  <Qd>, <Rt>`	Encoded as Q = 1
`VDUP<c><q>.<size>  <Dd>, <Rt>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VDUP` instruction must be unconditional.

`<size>` The data size for the elements of the destination vector. It must be one of:

- `8` encoded as [b,e] = 0b10
- `16` encoded as [b,e] = 0b01
- `32` encoded as [b,e] = 0b00.

`<Qd>` The destination vector for a quadword operation.

`<Dd>` The destination vector for a doubleword operation.

`<Rt>` The ARM source register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    scalar = R[t]<esize-1:0>;
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VDUP<c><q>.<size>  <Qd>, <Rt>`	Q = 1 としてエンコードされる
`VDUP<c><q>.<size>  <Dd>, <Rt>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。ARM `VDUP` は常に無条件命令です。

`<size>` デスティネーションベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` [b,e] = 0b10 としてエンコードされます。
- `16` [b,e] = 0b01 としてエンコードされます。
- `32` [b,e] = 0b00 としてエンコードされます。

`<Qd>` クワッドワード演算の場合のデスティネーションベクタ

`<Dd>` ダブルワード演算の場合のデスティネーションベクタ

`<Rt>` ARM ソースレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    scalar = R[t]<esize-1:0>;
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.304 VEOR

Vector Bitwise Exclusive OR performs a bitwise Exclusive OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Encoding T1 / A1 Advanced SIMD

```
VEOR<c> <Qd>, <Qn>, <Qm>
VEOR<c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.304 VEOR

VEOR（ベクタビット単位排他的論理和）は、2 つのレジスタの間でビット単位の排他的論理和演算を実行し、結果をデスティネーションレジスタに配置します。オペランドレジスタと結果レジスタには、クワッドワードまたはダブルワードを使用できます。オペランドと結果は、すべて同じサイズの必要があります。

エンコード T1/A1 アドバンスト SIMD

```
VEOR<c> <Qd>, <Qn>, <Qm>
VEOR<c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VEOR<c><q>{.<dt>} {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VEOR<c><q>{.<dt>} {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM `VEOR` instruction must be unconditional.
`<dt>`	An optional data type. It is ignored by assemblers, and does not affect the encoding.
`<Qd>, <Qn>, <Qm>`	The destination vector and the operand vectors, for a quadword operation.
`<Dd>, <Dn>, <Dm>`	The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] EOR D[m+r];
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VEOR<c><q>{.<dt>} {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VEOR<c><q>{.<dt>} {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VEOR` は常に無条件命令です。
`<dt>`	オプションのデータ型。アセンブラによって無視され、エンコードに影響を与えません。
`<Qd>, <Qn>, <Qm>`	クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Dd>, <Dn>, <Dm>`	ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] EOR D[m+r];
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.305 VEXT

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector. See Figure A8-1 for an example.

The elements of the vectors are treated as being 8-bit bitfields. There is no distinction between data types.

Figure A8-1 Operation of doubleword VEXT for imm = 3

Encoding T1 / A1 Advanced SIMD

```
VEXT<c>.8 <Qd>, <Qn>, <Qm>, #<imm>
VEXT<c>.8 <Dd>, <Dn>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	1	1	Vn				Vd				imm4				N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	1	1	Vn				Vd				imm4				N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1');  position = 8 * UInt(imm4);
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.305 VEXT

VEXT（ベクタ抽出）は、2 番目のオペランドベクタの最下位と最初のオペランドベクタの最上位からエレメントを抽出して連結し、結果をデスティネーションベクタに配置します。一例を図 A8-1 に示します。

ベクタのエレメントは、8 ビットのビットフィールドとして扱われます。これらにはデータ型の区別はありません。

図 A8-1 imm = 3 の場合のダブルワード VEXT の動作

エンコード T1/A1 アドバンスト SIMD

```
VEXT<c>.8 <Qd>, <Qn>, <Qm>, #<imm>
VEXT<c>.8 <Dd>, <Dn>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	1	1	Vn				Vd				imm4				N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	1	1	Vn				Vd				imm4				N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1');  position = 8 * UInt(imm4);
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VEXT<c><q>.<size> {<Qd>,} <Qn>, <Qm>, #<imm>`	Encoded as Q = 1
`VEXT<c><q>.<size> {<Dd>,} <Dn>, <Dm>, #<imm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VEXT` instruction must be unconditional.

`<size>` Size of the operation. The value can be:

- 8, 16, or 32 for doubleword operations
- 8, 16, 32, or 64 for quadword operations.

If the value is 16, 32, or 64, the syntax is a pseudo-instruction for a `VEXT` instruction specifying the equivalent number of bytes. The following examples show how an assembler treats values greater than 8:

`VEXT.16 D0,D1,#x` is treated as `VEXT.8 D0,D1,#(x*2)`

`VEXT.32 D0,D1,#x` is treated as `VEXT.8 D0,D1,#(x*4)`

`VEXT.64 Q0,Q1,#x` is treated as `VEXT.8 Q0,Q1,#(x*8)`.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

`<imm>` The location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0-7 for a doubleword operation or 0-15 for a quadword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        Q[d] = (Q[m]:Q[n])<position+127:position>;
    else
        D[d] = (D[m]:D[n])<position+63:position>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VEXT<c><q>.<size> {<Qd>,} <Qn>, <Qm>, #<imm>`	Q = 1 としてエンコードされる
`VEXT<c><q>.<size> {<Dd>,} <Dn>, <Dm>, #<imm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VEXT` は常に無条件命令です。

`<size>` 演算のサイズ。次の値を指定できます。

- 8、16、または 32（ダブルワード演算の場合）。
- 8、16、32、または 64（クワッドワード演算の場合）。

値が 16、32、または 64 の場合、構文は、同じ数のバイトを指定する `VEXT` 命令の擬似命令になります。8 より大きい値は、アセンブラで次の例のように扱われます。

`VEXT.16 D0,D1,#x`は、`VEXT.8 D0,D1,#(x*2)` として扱われます。

`VEXT.32 D0,D1,#x`は、`VEXT.8 D0,D1,#(x*4)` として扱われます。

`VEXT.64 Q0,Q1,#x`は、`VEXT.8 Q0,Q1,#(x*8)` として扱われます。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<imm>` オペランドの連結における抽出結果の位置。最下位からのバイト数として指定し、ダブルワード演算の場合は 0 ～ 7、クワッドワード演算の場合は 0 ～ 15 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        Q[d] = (Q[m]:Q[n])<position+127:position>;
    else
        D[d] = (D[m]:D[n])<position+63:position>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.306 VHADD, VHSUB

Vector Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated (for rounded results see *VRHADD* on page A8-734).

Vector Halving Subtract subtracts the elements of the second operand from the corresponding elements of the first operand, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated (there is no rounding version).

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers
- 8-bit, 16-bit, or 32-bit unsigned integers.

Encoding T1 / A1 Advanced SIMD

```
VH<op><c> <Qd>, <Qn>, <Qm>
VH<op><c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	op	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	op	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.306 VHADD、VHSUB

VHADD（ベクタ加算半減）は、整数の 2 つのベクタの対応するエレメントを加算し、各結果を右に 1 ビットシフトし、最終結果をデスティネーションベクタに配置します。半減演算の結果は切り捨てられます（丸められる結果については、*VRHADD* P. A8- 734 を参照して下さい）。

VHSUB（ベクタ減算半減）は、2 番目のオペランドのエレメントを最初のオペランドの対応するエレメントから減算し、各結果を右に 1 ビットシフトし、最終結果をデスティネーションベクタに配置します。半減演算の結果は切り捨てられます（丸めバージョンはありません）。

オペランドと結果のエレメントのデータ型はすべて同じで、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットの符号付き整数
- 8 ビット、16 ビット、または 32 ビットの符号なし整数

エンコード T1/A1 アドバンスト SIMD

```
VH<op><c> <Qd>, <Qn>, <Qm>
VH<op><c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	op	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	op	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VH<op><c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VH<op><c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<op>` Must be one of:
- `ADD` encoded as op = 0
- `SUB` encoded as op = 1.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VHADD` or `VHSUB` instruction must be unconditional.

`<dt>` The data type for the elements of the vectors. It must be one of:
- `S8` encoded as size = 0b00, U = 0
- `S16` encoded as size = 0b01, U = 0
- `S32` encoded as size = 0b10, U = 0
- `U8` encoded as size = 0b00, U = 1
- `U16` encoded as size = 0b01, U = 1
- `U32` encoded as size = 0b10, U = 1.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if add then op1+op2 else op1-op2;
            Elem[D[d+r],e,esize] = result<esize:1>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VH<op><c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VH<op><c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<op>` 次のいずれかを使用できます。
- `ADD` op = 0 としてエンコードされます。
- `SUB` op = 1 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VHADD` および `VHSUB` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。
- `S8` size = 0b00、U = 0 としてエンコードされます。
- `S16` size = 0b01、U = 0 としてエンコードされます。
- `S32` size = 0b10、U = 0 としてエンコードされます。
- `U8` size = 0b00、U = 1 としてエンコードされます。
- `U16` size = 0b01、U = 1 としてエンコードされます。
- `U32` size = 0b10、U = 1 としてエンコードされます。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if add then op1+op2 else op1-op2;
            Elem[D[d+r],e,esize] = result<esize:1>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.307 VLD1 (multiple single elements)

This instruction loads elements from memory into one, two, three, or four registers, without de-interleaving. Every element of each register is loaded. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VLD1<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD1<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	1	0	Rn				Vd				type				size		align		Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	1	0	Rn				Vd				type				size		align		Rm			

```
case type of
    when '0111'
        regs = 1;  if align<1> == '1' then UNDEFINED;
    when '1010'
        regs = 2;  if align == '11' then UNDEFINED;
    when '0110'
        regs = 3;  if align<1> == '1' then UNDEFINED;
    when '0010'
        regs = 4;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d+regs > 32 then UNPREDICTABLE;
```

Related encodings See *Advanced SIMD element or structure load/store instructions* on page A7-27

Assembler syntax

```
VLD1<c><q>.<size>  <list>,[<Rn>{@<align>}]          Rm = '1111'
VLD1<c><q>.<size>  <list>, [<Rn>{@<align>}]!        Rm = '1101'
VLD1<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>   Rm = other values
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VLD1` instruction must be unconditional.

`<size>` The data size. It must be one of:

- 8 encoded as size = 0b00
- 16 encoded as size = 0b01

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.307 VLD1（複数の単一エレメントのロード）

VLD1（複数の単一エレメントのロード）は、エレメントをメモリから 1 つ、2 つ、3 つ、または 4 つのレジスタに、インターリーブ解除なしでロードします。各レジスタのすべてのエレメントがロードされます。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VLD1<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD1<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	1	0	Rn				Vd				type				size		align		Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	1	0	Rn				Vd				type				size		align		Rm			

```
case type of
    when '0111'
        regs = 1;  if align<1> == '1' then UNDEFINED;
    when '1010'
        regs = 2;  if align == '11' then UNDEFINED;
    when '0110'
        regs = 3;  if align<1> == '1' then UNDEFINED;
    when '0010'
        regs = 4;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d+regs > 32 then UNPREDICTABLE;
```

関連エンコード *アドバンスト SIMD のエレメントまたは構造ロード / ストア命令* P. A7-27 を参照して下さい。

アセンブラ構文

```
VLD1<c><q>.<size>  <list>,[<Rn>{@<align>}]          Rm = '1111'
VLD1<c><q>.<size>  <list>, [<Rn>{@<align>}]!        Rm = '1101'
VLD1<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>   Rm = その他の値
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VLD1` は常に無条件命令です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

32 encoded as size = 0b10

64 encoded as size = 0b11.

<list> The list of registers to load. It must be one of:

{<Dd>} encoded as D:Vd = <Dd>, type = 0b0111

{<Dd>, <Dd+1>} encoded as D:Vd = <Dd>, type = 0b1010

{<Dd>, <Dd+1>, <Dd+2>} encoded as D:Vd = <Dd>, type = 0b0110

{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>} encoded as D:Vd = <Dd>, type = 0b0010.

<Rn> Contains the base address for the access.

<align> The alignment. It can be one of:

64 8-byte alignment, encoded as align = 0b01.

128 16-byte alignment, available only if <list> contains two or four registers, encoded as align = 0b10.

256 32-byte alignment, available only if <list> contains four registers, encoded as align = 0b11.

omitted Standard alignment, see *Unaligned data access* on page A3-5. Encoded as align = 0b00.

! If present, specifies writeback.

<Rm> Contains an address offset applied after the access.

For more information about <Rn>, !, and <Rm>, see *Advanced SIMD addressing mode* on page A7-30.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 8*regs);
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = MemU[address,ebytes];
            address = address + ebytes;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<size> データサイズ。次のいずれかを使用できます。

8 size = 0b00 としてエンコードされます。

16 size = 0b01 としてエンコードされます。

32 size = 0b10 としてエンコードされます。

64 size = 0b11 としてエンコードされます。

<list> ロードするレジスタのリスト。次のいずれかを使用できます。

{<Dd>} D:Vd = <Dd>、type = 0b0111 としてエンコードされます。

{<Dd>, <Dd+1>} D:Vd = <Dd>、type = 0b1010 としてエンコードされます。

{<Dd>, <Dd+1>, <Dd+2>} D:Vd = <Dd>、type = 0b0110 としてエンコードされます。

{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>} D:Vd = <Dd>、type = 0b0010 としてエンコードされます。

<Rn> アクセス用のベースアドレス。

<align> アライメント。次のいずれかを使用できます。

64 8 バイトアライメント。align = 0b01 としてエンコードされます。

128 16 バイトアライメント。<list> に 2 つまたは 4 つのレジスタが含まれる場合のみ使用できます。align = 0b10 としてエンコードされます。

256 32 バイトアライメント。<list> に 4 つのレジスタが含まれる場合のみ使用できます。align = 0b11 としてエンコードされます。

省略 標準的なアライメント。詳細については、*アンアライインド データアクセス* P. A3-5 を参照して下さい。align = 0b00 としてエンコードされます。

! 存在する場合、ライトバックが指定されます。

<Rm> アクセス後に適用されるアドレスオフセット

<Rn>、!、および <Rm> の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 8*regs);
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = MemU[address,ebytes];
            address = address + ebytes;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.308 VLD1 (single element to one lane)

This instruction loads one element from memory into one element of a register. Elements of the register that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VLD1<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD1<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				size		0	0	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				size		0	0	index_align				Rm			

```
if size == '11' then SEE VLD1 (single element to all lanes);
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  alignment = 1;
    when '01'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        alignment = if index_align<0> == '0' then 1 else 2;
    when '10'
        if index_align<2> != '0' then UNDEFINED;
        if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
```

Assembler syntax

`VLD1<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VLD1<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VLD1<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VLD1` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.308 VLD1（1 つのレーンへの単一エレメントのロード）

VLD1（1 つのレーンへの単一エレメントのロード）は、1 つのエレメントをメモリからレジスタの 1 つのエレメントにロードします。そのレジスタの、ロード対象でないエレメントは変更されません。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VLD1<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD1<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				size		0	0	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				size		0	0	index_align				Rm			

```
if size == '11' then SEE VLD1 (single element to all lanes);
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  alignment = 1;
    when '01'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        alignment = if index_align<0> == '0' then 1 else 2;
    when '10'
        if index_align<2> != '0' then UNDEFINED;
        if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
```

アセンブラ構文

`VLD1<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VLD1<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VLD1<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VLD1` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` The register containing the element to load. It must be `{<Dd[x]>}`. The register `<Dd>` is encoded in `D:Vd`.

`<Rn>` Contains the base address for the access.

`<align>` The alignment. It can be one of:

- `16` 2-byte alignment, available only if `<size>` is 16
- `32` 4-byte alignment, available only if `<size>` is 32
- **omitted** Standard alignment, see *Unaligned data access* on page A3-5.

`!` If present, specifies writeback.

`<Rm>` Contains an address offset applied after the access.

For more information about `<Rn>`, `!`, and `<Rm>`, see *Advanced SIMD addressing mode* on page A7-30.

Table A8-5 shows the encoding of index and alignment for the different `<size>` values.

Table A8-5 Encoding of index and alignment

	`<size>` == 8	`<size>` == 16	`<size>` == 32
Index	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
`<align>` omitted	index_align[0] = 0	index_align[1:0] = '00'	index_align[2:0] = '000'
`<align>` == 16	-	index_align[1:0] = '01'	-
`<align>` == 32	-	-	index_align[2:0] = '011'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else ebytes);
    Elem[D[d],index,esize] = MemU[address,ebytes];
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` ロードするエレメントを含むレジスタ。`{<Dd[x]>}` にする必要があります。レジスタ `<Dd>` は、`D:Vd` にエンコードされます。

`<Rn>` アクセス用のベースアドレス。

`<align>` アライメント。次のいずれかを使用できます。

- `16` 2 バイトアライメント。`<size>` が 16 の場合にのみ使用できます。
- `32` 4 バイトアライメント。`<size>` が 32 の場合にのみ使用できます。
- **省略** 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。

`!` 存在する場合、ライトバックが指定されます。

`<Rm>` アクセス後に適用されるアドレスオフセット

`<Rn>`、`!`、および `<Rm>` の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

さまざまな `<size>` 値でのインデクスとアライメントのエンコードを表 A8-5 に示します。

表 A8-5 インデクスとアライメントのエンコード

	`<size>` == 8	`<size>` == 16	`<size>` == 32
インデクス	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
`<align>` 省略	index_align[0] = 0	index_align[1:0] = '00'	index_align[2:0] = '000'
`<align>` == 16	–	index_align[1:0] = '01'	–
`<align>` == 32	–	–	index_align[2:0] = '011'

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else ebytes);
    Elem[D[d],index,esize] = MemU[address,ebytes];
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.309 VLD1 (single element to all lanes)

This instruction loads one element from memory into every element of one or two vectors. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VLD1<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD1<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	1	0	0	size		T	a	Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	1	0	0	size		T	a	Rm			

```
if size == '11' || (size == '00' && a == '1') then UNDEFINED;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;  regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d+regs > 32 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.309 VLD1（全レーンへの単一エレメントのロード）

VLD1（全レーンへの単一エレメントのロード）は、1 つのエレメントをメモリから 1 つまたは 2 つのベクタのすべてのエレメントにロードします。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VLD1<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD1<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	1	0	0	size		T	a	Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	1	0	0	size		T	a	Rm			

```
if size == '11' || (size == '00' && a == '1') then UNDEFINED;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;  regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d+regs > 32 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VLD1<c><q>.<size>  <list>, [<Rn>{@<align>}]                Rm = '1111'
VLD1<c><q>.<size>  <list>, [<Rn>{@<align>}]!               Rm = '1101'
VLD1<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>          Rm = other values
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VLD1` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

`<list>` The list of registers to load. It must be one of:

- `{<Dd[]>}` encoded as D:Vd = `<Dd>`, T = 0
- `{<Dd[]>, <Dd+1[]>}` encoded as D:Vd = `<Dd>`, T = 1.

`<Rn>` Contains the base address for the access.

`<align>` The alignment. It can be one of:

- `16` 2-byte alignment, available only if `<size>` is 16, encoded as a = 1.
- `32` 4-byte alignment, available only if `<size>` is 32, encoded as a = 1.
- **omitted** Standard alignment, see *Unaligned data access* on page A3-5. Encoded as a = 0.

`!` If present, specifies writeback.

`<Rm>` Contains an address offset applied after the access.

For more information about `<Rn>`, `!`, and `<Rm>`, see *Advanced SIMD addressing mode* on page A7-30.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else ebytes);
    replicated_element = Replicate(MemU[address,ebytes], elements);
    for r = 0 to regs-1
        D[d+r] = replicated_element;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VLD1<c><q>.<size>  <list>, [<Rn>{@<align>}]                Rm = '1111'
VLD1<c><q>.<size>  <list>, [<Rn>{@<align>}]!               Rm = '1101'
VLD1<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>          Rm = その他の値
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VLD1` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

`<list>` ロードするレジスタのリスト。次のいずれかを使用できます。

- `{<Dd[]>}` D:Vd = `<Dd>`、T = 0 としてエンコードされます。
- `{<Dd[]>, <Dd+1[]>}` D:Vd = `<Dd>`、T = 1 としてエンコードされます。

`<Rn>` アクセス用のベースアドレス。

`<align>` アライメント。次のいずれかを使用できます。

- `16` 2 バイトアライメント。`<size>` が 16 の場合にのみ使用できます。a = 1 としてエンコードされます。
- `32` 4 バイトアライメント。`<size>` が 32 の場合にのみ使用できます。a = 1 としてエンコードされます。
- **省略** 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。a = 0 としてエンコードされます。

`!` 存在する場合、ライトバックが指定されます。

`<Rm>` アクセス後に適用されるアドレスオフセット

`<Rn>`、`!`、および `<Rm>` の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else ebytes);
    replicated_element = Replicate(MemU[address,ebytes], elements);
    for r = 0 to regs-1
        D[d+r] = replicated_element;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.310 VLD2 (multiple 2-element structures)

This instruction loads multiple 2-element structures from memory into two or four registers, with de-interleaving. For more information, see *Element and structure load/store instructions* on page A4-27. Every element of each register is loaded. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VLD2<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD2<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	1	0	Rn				Vd				type				size		align		Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	1	0	Rn				Vd				type				size		align		Rm			

```
if size == '11' then UNDEFINED;
case type of
    when '1000'
        regs = 1;  inc = 1;  if align == '11' then UNDEFINED;
    when '1001'
        regs = 1;  inc = 2;  if align == '11' then UNDEFINED;
    when '0011'
        regs = 2;  inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d2+regs > 32 then UNPREDICTABLE;
```

Related encodings See *Advanced SIMD element or structure load/store instructions* on page A7-27

Assembler syntax

`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VLD2` instruction must be unconditional.

`<size>` The data size. It must be one of:

- 8 encoded as size = 0b00
- 16 encoded as size = 0b01

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.310 VLD2（複数の 2 エレメント構造のロード）

VLD2（複数の 2 エレメント構造のロード）は、複数の 2 エレメント構造をメモリから 2 つまたは 4 つのレジスタに、インターリーブ解除を行ってロードします。詳細については、*エレメントおよび構造のロード / ストア命令* P. A4- 27 を参照して下さい。各レジスタのすべてのエレメントがロードされます。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7- 30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VLD2<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD2<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	1	0	Rn				Vd				type				size		align		Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	1	0	Rn				Vd				type				size		align		Rm			

```
if size == '11' then UNDEFINED;
case type of
    when '1000'
        regs = 1;  inc = 1;  if align == '11' then UNDEFINED;
    when '1001'
        regs = 1;  inc = 2;  if align == '11' then UNDEFINED;
    when '0011'
        regs = 2;  inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d2+regs > 32 then UNPREDICTABLE;
```

関連エンコード *アドバンスト SIMD のエレメントまたは構造ロード / ストア命令* P. A7- 27 を参照して下さい。

アセンブラ構文

`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。ARM `VLD2` は常に無条件命令です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

32 encoded as size = 0b10.

<list> The list of registers to load. It must be one of:

{<Dd>, <Dd+1>} encoded as D:Vd = <Dd>, type = 0b1000

{<Dd>, <Dd+2>} encoded as D:Vd = <Dd>, type = 0b1001

{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>}
encoded as D:Vd = <Dd>, type = 0b0011.

<Rn> Contains the base address for the access.

<align> The alignment. It can be one of:

64 8-byte alignment, encoded as align = 0b01.

128 16-byte alignment, encoded as align = 0b10.

256 32-byte alignment, available only if <list> contains four registers. Encoded as align = 0b11

omitted Standard alignment, see *Unaligned data access* on page A3-5. Encoded as align = 0b00.

! If present, specifies writeback.

<Rm> Contains an address offset applied after the access.

For more information about <Rn>, !, and <Rm>, see *Advanced SIMD addressing mode* on page A7-30.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 16*regs);
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = MemU[address,ebytes];
            Elem[D[d2+r],e,esize] = MemU[address+ebytes,ebytes];
            address = address + 2*ebytes;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<size> データサイズ。次のいずれかを使用できます。

8 size = 0b00 としてエンコードされます。

16 size = 0b01 としてエンコードされます。

32 size = 0b10 としてエンコードされます。

<list> ロードするレジスタのリスト。次のいずれかを使用できます。

{<Dd>, <Dd+1>} D:Vd = <Dd>、type = 0b1000 としてエンコードされます。

{<Dd>, <Dd+2>} D:Vd = <Dd>、type = 0b1001 としてエンコードされます。

{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>}
D:Vd = <Dd>、type = 0b0011 としてエンコードされます。

<Rn> アクセス用のベースアドレス。

<align> アライメント。次のいずれかを使用できます。

64 8 バイトアライメント。align = 0b01 としてエンコードされます。

128 16 バイトアライメント。align = 0b10 としてエンコードされます。

256 32 バイトアライメント。<list> に 4 つのレジスタが含まれる場合にのみ使用できます。align = 0b11 としてエンコードされます。

省略 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。align = 0b00 としてエンコードされます。

! 存在する場合、ライトバックが指定されます。

<Rm> アクセス後に適用されるアドレスオフセット

<Rn>、!、および <Rm> の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 16*regs);
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = MemU[address,ebytes];
            Elem[D[d2+r],e,esize] = MemU[address+ebytes,ebytes];
            address = address + 2*ebytes;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.311 VLD2 (single 2-element structure to one lane)

This instruction loads one 2-element structure from memory into corresponding elements of two registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VLD2<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD2<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				size		0	1	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				size		0	1	index_align				Rm			

```
if size == '11' then SEE VLD2 (single 2-element structure to all lanes);
case size of
    when '00'
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 2;
    when '01'
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '10'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d2 > 31 then UNPREDICTABLE;
```

Assembler syntax

`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VLD2` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.311 VLD2（1 つのレーンへの単一の 2 エレメント構造のロード）

VLD2（1 つのレーンへの単一の 2 エレメント構造のロード）は、1 つの 2 エレメント構造をメモリから、2 つのレジスタの対応するエレメントにロードします。レジスタの、ロード対象でないエレメントは変更されません。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VLD2<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD2<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				size		0	1	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				size		0	1	index_align				Rm			

```
if size == '11' then SEE VLD2 (single 2-element structure to all lanes);
case size of
    when '00'
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 2;
    when '01'
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '10'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d2 > 31 then UNPREDICTABLE;
```

アセンブラ構文

`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VLD2` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<list> The registers containing the structure. Encoded with D:Vd = <Dd>. It must be one of:

{<Dd[x]>, <Dd+1[x]>} Single-spaced registers, see Table A8-6.

{<Dd[x]>, <Dd+2[x]>} Double-spaced registers, see Table A8-6.
This is not available if <size> == 8.

<Rn> Contains the base address for the access.

<align> The alignment. It can be one of:

16 2-byte alignment, available only if <size> is 8

32 4-byte alignment, available only if <size> is 16

64 8-byte alignment, available only if <size> is 32

omitted Standard alignment, see *Unaligned data access* on page A3-5.

! If present, specifies writeback.

<Rm> Contains an address offset applied after the access.

For more information about <Rn>, !, and <Rm> see *Advanced SIMD addressing mode* on page A7-30.

Table A8-6 Encoding of index, alignment, and register spacing

	<size> == 8	<size> == 16	<size> == 32
Index	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
Single-spacing	-	index_align[1] = 0	index_align[2] = 0
Double-spacing	-	index_align[1] = 1	index_align[2] = 1
<align> omitted	index_align[0] = 0	index_align[0] = 0	index_align[1:0] = '00'
<align> == 16	index_align[0] = 1	-	-
<align> == 32	-	index_align[0] = 1	-
<align> == 64	-	-	index_align[1:0] = '01'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 2*ebytes);
    Elem[D[d],index,esize] = MemU[address,ebytes];
    Elem[D[d2],index,esize] = MemU[address+ebytes,ebytes];
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<list> 構造を含むレジスタ。D:Vd = <Dd> としてエンコードされます。次のいずれかを使用できます。

{<Dd[x]>, <Dd+1[x]>} シングルスペース レジスタ（表 A8-6 参照）。

{<Dd[x]>, <Dd+2[x]>} ダブルスペース レジスタ（表 A8-6 参照）。
<size> == 8 の場合には使用できません。

<Rn> アクセス用のベースアドレス。

<align> アライメント。次のいずれかを使用できます。

16 2 バイトアライメント。<size> が 8 の場合にのみ使用できます。

32 4 バイトアライメント。<size> が 16 の場合にのみ使用できます。

64 8 バイトアライメント。<size> が 32 の場合にのみ使用できます。

省略 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。

! 存在する場合、ライトバックが指定されます。

<Rm> アクセス後に適用されるアドレスオフセット

<Rn>、!、および <Rm> の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

表 A8-6 インデクス、アライメント、およびレジスタスペーシングのエンコード

	<size> == 8	<size> == 16	<size> == 32
インデクス	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
シングルスペーシング	–	index_align[1] = 0	index_align[2] = 0
ダブルスペーシング	–	index_align[1] = 1	index_align[2] = 1
<align> 省略	index_align[0] = 0	index_align[0] = 0	index_align[1:0] = '00'
<align> == 16	index_align[0] = 1	–	–
<align> == 32	–	index_align[0] = 1	–
<align> == 64	–	–	index_align[1:0] = '01'

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 2*ebytes);
    Elem[D[d],index,esize] = MemU[address,ebytes];
    Elem[D[d2],index,esize] = MemU[address+ebytes,ebytes];
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.312 VLD2 (single 2-element structure to all lanes)

This instruction loads one 2-element structure from memory into all lanes of two registers. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VLD2<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD2<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	1	0	1	size		T	a	Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	1	0	1	size		T	a	Rm			

```
if size == '11' then UNDEFINED;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
alignment = if a == '0' then 1 else 2*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d2 > 31 then UNPREDICTABLE;
```

Assembler syntax

`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VLD2` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

`<list>` The registers containing the structure. It must be one of:

- `{<Dd[]>, <Dd+1[]>}` single-spaced register transfer, encoded as D:Vd = `<Dd>`, T = 0
- `{<Dd[]>, <Dd+2[]>}` double-spaced register transfer, encoded as D:Vd = `<Dd>`, T = 1.

`<Rn>` Contains the base address for the access.

`<align>` The alignment. It can be one of:

- `16` 2-byte alignment, available only if `<size>` is 8, encoded as a = 1
- `32` 4-byte alignment, available only if `<size>` is 16, encoded as a = 1

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.312 VLD2（全レーンへの単一の 2 エレメント構造のロード）

VLD2（全レーンへの単一の 2 エレメント構造のロード）は、1 つの 2 エレメント構造をメモリから、2 つのレジスタの全レーンにロードします。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VLD2<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD2<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	1	0	1	size		T	a	Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	1	0	1	size		T	a	Rm			

```
if size == '11' then UNDEFINED;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
alignment = if a == '0' then 1 else 2*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d2 > 31 then UNPREDICTABLE;
```

アセンブラ構文

`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VLD2<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VLD2` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

`<list>` 構造を含むレジスタ。次のいずれかを使用できます。

- `{<Dd[]>, <Dd+1[]>}` シングルスペース レジスタ転送。D:Vd = `<Dd>`、T = 0 としてエンコードされます。
- `{<Dd[]>, <Dd+2[]>}` ダブルスペース レジスタ転送。D:Vd = `<Dd>`、T = 1 としてエンコードされます。

`<Rn>` アクセス用のベースアドレス。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

64	8-byte alignment, available only if <size> is 32, encoded as a = 1
omitted	Standard alignment, see *Unaligned data access* on page A3-5. Encoded as a = 0.

!	If present, specifies writeback.
<Rm>	Contains an address offset applied after the access.

For more information about <Rn>, !, and <Rm>, see *Advanced SIMD addressing mode* on page A7-30.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 2*ebytes);
    D[d] = Replicate(MemU[address,ebytes], elements);
    D[d2] = Replicate(MemU[address+ebytes,ebytes], elements);
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<align>	アライメント。次のいずれかを使用できます。
16	2 バイトアライメント。<size> が 8 の場合にのみ使用できます。a = 1 としてエンコードされます。
32	4 バイトアライメント。<size> が 16 の場合にのみ使用できます。a = 1 としてエンコードされます。
64	8 バイトアライメント。<size> が 32 の場合にのみ使用できます。a = 1 としてエンコードされます。
省略	標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。a = 0 としてエンコードされます。

!	存在する場合、ライトバックが指定されます。
<Rm>	アクセス後に適用されるアドレスオフセット

<Rn>、!、および <Rm> の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 2*ebytes);
    D[d] = Replicate(MemU[address,ebytes], elements);
    D[d2] = Replicate(MemU[address+ebytes,ebytes], elements);
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.313 VLD3 (multiple 3-element structures)

This instruction loads multiple 3-element structures from memory into three registers, with de-interleaving. For more information, see *Element and structure load/store instructions* on page A4-27. Every element of each register is loaded. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VLD3<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD3<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15 14 13 12	11 10 9 8	7	6	5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 1	1 0 0 1	0	D	1 0	Rn	Vd	type	size	align	Rm

31 30 29 28	27 26 25 24	23	22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 1	0 1 0 0	0	D	1 0	Rn	Vd	type	size	align	Rm

```
if size == '11' || align<1> == '1' then UNDEFINED;
case type of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d3 > 31 then UNPREDICTABLE;
```

Related encodings See *Advanced SIMD element or structure load/store instructions* on page A7-27

Assembler syntax

```
VLD3<c><q>.<size>  <list>, [<Rn>{@<align>}]          Rm = '1111'
VLD3<c><q>.<size>  <list>, [<Rn>{@<align>}]!         Rm = '1101'
VLD3<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>    Rm = other values
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VLD3` instruction must be unconditional.

`<size>` The data size. It must be one of:

- 8 encoded as size = 0b00
- 16 encoded as size = 0b01
- 32 encoded as size = 0b10.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.313 VLD3（複数の 3 エレメント構造のロード）

VLD3（複数の 3 エレメント構造のロード）は、複数の 3 エレメント構造をメモリから 3 つのレジスタに、インターリーブ解除を行ってロードします。詳細については、*エレメントおよび構造のロード／ストア命令* P. A4-27 を参照して下さい。各レジスタのすべてのエレメントがロードされます。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VLD3<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD3<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15 14 13 12	11 10 9 8	7	6	5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 1	1 0 0 1	0	D	1 0	Rn	Vd	type	size	align	Rm

31 30 29 28	27 26 25 24	23	22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 1	0 1 0 0	0	D	1 0	Rn	Vd	type	size	align	Rm

```
if size == '11' || align<1> == '1' then UNDEFINED;
case type of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d3 > 31 then UNPREDICTABLE;
```

関連エンコード *アドバンスト SIMD のエレメントまたは構造ロード／ストア命令* P. A7-27 を参照して下さい。

アセンブラ構文

```
VLD3<c><q>.<size>  <list>, [<Rn>{@<align>}]          Rm = '1111'
VLD3<c><q>.<size>  <list>, [<Rn>{@<align>}]!         Rm = '1101'
VLD3<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>    Rm = その他の値
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VLD3` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- 8 size = 0b00 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` The list of registers to load. It must be one of:

`{<Dd>, <Dd+1>, <Dd+2>}`
encoded as D:Vd = `<Dd>`, type = 0b0100

`{<Dd>, <Dd+2>, <Dd+4>}`
encoded as D:Vd = `<Dd>`, type = 0b0101.

`<Rn>` Contains the base address for the access.

`<align>` The alignment. It can be:

`64` 8-byte alignment, encoded as align = 0b01.

omitted Standard alignment, see *Unaligned data access* on page A3-5. Encoded as align = 0b00.

`!` If present, specifies writeback.

`<Rm>` Contains an address offset applied after the access.

For more information about `<Rn>`, `!`, and `<Rm>`, see *Advanced SIMD addressing mode* on page A7-30.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 24);
    for e = 0 to elements-1
        Elem[D[d],e,esize] = MemU[address,ebytes];
        Elem[D[d2],e,esize] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e,esize] = MemU[address+2*ebytes,ebytes];
        address = address + 3*ebytes;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`16` size = 0b01 としてエンコードされます。

`32` size = 0b10 としてエンコードされます。

`<list>` ロードするレジスタのリスト。次のいずれかを使用できます。

`{<Dd>, <Dd+1>, <Dd+2>}`
D:Vd = `<Dd>`、type = 0b0100 としてエンコードされます。

`{<Dd>, <Dd+2>, <Dd+4>}`
D:Vd = `<Dd>`、type = 0b0101 としてエンコードされます。

`<Rn>` アクセス用のベースアドレス。

`<align>` アライメント。次の値を使用できます。

`64` 8 バイトアライメント。align = 0b01 としてエンコードされます。

省略 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。align = 0b00 としてエンコードされます。

`!` 存在する場合、ライトバックが指定されます。

`<Rm>` アクセス後に適用されるアドレスオフセット

`<Rn>`、`!`、および `<Rm>` の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 24);
    for e = 0 to elements-1
        Elem[D[d],e,esize] = MemU[address,ebytes];
        Elem[D[d2],e,esize] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e,esize] = MemU[address+2*ebytes,ebytes];
        address = address + 3*ebytes;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.314 VLD3 (single 3-element structure to one lane)

This instruction loads one 3-element structure from memory into corresponding elements of three registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VLD3<c>.<size> <list>, [<Rn>]{!}
VLD3<c>.<size> <list>, [<Rn>], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				size		1	0	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				size		1	0	index_align				Rm			

```
if size == '11' then SEE VLD3 (single 3-element structure to all lanes);
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  inc = 1;
    when '01'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
    when '10'
        if index_align<1:0> != '00' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d3 > 31 then UNPREDICTABLE;
```

Assembler syntax

`VLD3<c><q>.<size> <list>, [<Rn>]`	Rm = '1111'
`VLD3<c><q>.<size> <list>, [<Rn>]!`	Rm = '1101'
`VLD3<c><q>.<size> <list>, [<Rn>], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VLD3` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.314 VLD3（1 つのレーンへの単一の 3 エレメント構造のロード）

VLD3（1 つのレーンへの単一の 3 エレメント構造のロード）は、1 つの 3 エレメント構造をメモリから、3 つのレジスタの対応するエレメントにロードします。レジスタの、ロード対象でないエレメントは変更されません。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VLD3<c>.<size> <list>, [<Rn>]{!}
VLD3<c>.<size> <list>, [<Rn>], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				size		1	0	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				size		1	0	index_align				Rm			

```
if size == '11' then SEE VLD3 (single 3-element structure to all lanes);
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  inc = 1;
    when '01'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
    when '10'
        if index_align<1:0> != '00' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d3 > 31 then UNPREDICTABLE;
```

アセンブラ構文

`VLD3<c><q>.<size> <list>, [<Rn>]`	Rm = '1111'
`VLD3<c><q>.<size> <list>, [<Rn>]!`	Rm = '1101'
`VLD3<c><q>.<size> <list>, [<Rn>], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VLD3` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` The registers containing the structure. Encoded with D:Vd = `<Dd>`. It must be one of:

`{<Dd[x]>, <Dd+1[x]>, <Dd+2[x]>}`
Single-spaced registers, see Table A8-7.

`{<Dd[x]>, <Dd+2[x]>, <Dd+4[x]>}`
Double-spaced registers, see Table A8-7. This is not available if `<size>` == 8.

`<Rn>` Contains the base address for the access.

`!` If present, specifies writeback.

`<Rm>` Contains an address offset applied after the access.

For more information about `<Rn>`, `!`, and `<Rm>`, see *Advanced SIMD addressing mode* on page A7-30.

Table A8-7 Encoding of index and register spacing

	`<size>` == 8	`<size>` == 16	`<size>` == 32
Index	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
Single-spacing	index_align[0] = 0	index_align[1:0] = '00'	index_align[2:0] = '000'
Double-spacing	-	index_align[1:0] = '10'	index_align[2:0] = '100'

Alignment

Standard alignment rules apply, see *Unaligned data access* on page A3-5.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];
    if wback then R[n] = R[n] + (if register_index then R[m] else 3*ebytes);
    Elem[D[d],index,esize] = MemU[address,ebytes];
    Elem[D[d2],index,esize] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index,esize] = MemU[address+2*ebytes,ebytes];
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` 構造を含むレジスタ。D:Vd = `<Dd>` としてエンコードされます。次のいずれかを使用できます。

`{<Dd[x]>, <Dd+1[x]>, <Dd+2[x]>}`
シングルスペース レジスタ（表 A8-7 参照）。

`{<Dd[x]>, <Dd+2[x]>, <Dd+4[x]>}`
ダブルスペース レジスタ（表 A8-7 参照）。`<size>` == 8 の場合には使用できません。

`<Rn>` アクセス用のベースアドレス。

`!` 存在する場合、ライトバックが指定されます。

`<Rm>` アクセス後に適用されるアドレスオフセット

`<Rn>`、`!`、および `<Rm>` の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

表 A8-7 インデクスとレジスタスペーシングのエンコード

	`<size>` == 8	`<size>` == 16	`<size>` == 32
インデクス	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
シングルスペーシング	index_align[0] = 0	index_align[1:0] = ’00’	index_align[2:0] = ’000’
ダブルスペーシング	–	index_align[1:0] = ’10’	index_align[2:0] = ’100’

アライメント

標準的なアライメント規則が適用されます。詳細については、*アンアライインド データアクセス* P. A3-5 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];
    if wback then R[n] = R[n] + (if register_index then R[m] else 3*ebytes);
    Elem[D[d],index,esize] = MemU[address,ebytes];
    Elem[D[d2],index,esize] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index,esize] = MemU[address+2*ebytes,ebytes];
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.315 VLD3 (single 3-element structure to all lanes)

This instruction loads one 3-element structure from memory into all lanes of three registers. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VLD3<c>.<size> <list>, [<Rn>]{!}
VLD3<c>.<size> <list>, [<Rn>], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	1	1	0	size		T	a	Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	1	1	0	size		T	a	Rm			

```
if size == '11' || a == '1' then UNDEFINED;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d3 > 31 then UNPREDICTABLE;
```

Assembler syntax

`VLD3<c><q>.<size> <list>, [<Rn>]`	Rm = '1111'
`VLD3<c><q>.<size> <list>, [<Rn>]!`	Rm = '1101'
`VLD3<c><q>.<size> <list>, [<Rn>], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VLD3` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

`<list>` The registers containing the structures. It must be one of:

`{<Dd[]>, <Dd+1[]>, <Dd+2[]>}`
Single-spaced register transfer, encoded as D:Vd = `<Dd>`, T = 0.

`{<Dd[]>, <Dd+2[]>, <Dd+4[]>}`
Double-spaced register transfer, encoded as D:Vd = `<Dd>`, T = 1.

`<Rn>` Contains the base address for the access.

`!` If present, specifies writeback.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.315 VLD3（全レーンへの単一の 3 エレメント構造のロード）

この命令は、1 つの 3 エレメント構造をメモリから、3 つのレジスタの全レーンにロードします。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VLD3<c>.<size> <list>, [<Rn>]{!}
VLD3<c>.<size> <list>, [<Rn>], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	1	1	0	size		T	a	Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	1	1	0	size		T	a	Rm			

```
if size == '11' || a == '1' then UNDEFINED;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d3 > 31 then UNPREDICTABLE;
```

アセンブラ構文

`VLD3<c><q>.<size> <list>, [<Rn>]`	Rm = '1111'
`VLD3<c><q>.<size> <list>, [<Rn>]!`	Rm = '1101'
`VLD3<c><q>.<size> <list>, [<Rn>], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VLD3` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

`<list>` 構造を含むレジスタ。次のいずれかを使用できます。

`{<Dd[]>, <Dd+1[]>, <Dd+2[]>}`
シングルスペース レジスタ転送。D:Vd = `<Dd>`、T = 0 としてエンコードされます。

`{<Dd[]>, <Dd+2[]>, <Dd+4[]>}`
ダブルスペース レジスタ転送。D:Vd = `<Dd>`、T = 1 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<Rm> Contains an address offset applied after the access.

For more information about <Rn>, !, and <Rm>, see *Advanced SIMD addressing mode* on page A7-30.

Alignment

Standard alignment rules apply, see *Unaligned data access* on page A3-5.

The a bit must be encoded as 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];
    if wback then R[n] = R[n] + (if register_index then R[m] else 3*ebytes);
    D[d] = Replicate(MemU[address,ebytes], elements);
    D[d2] = Replicate(MemU[address+ebytes,ebytes], elements);
    D[d3] = Replicate(MemU[address+2*ebytes,ebytes], elements);
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

<Rn> アクセス用のベースアドレス。

! 存在する場合、ライトバックが指定されます。

<Rm> アクセス後に適用されるアドレスオフセット

<Rn>、!、および <Rm> の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

アライメント

標準的なアライメント規則が適用されます。詳細については、*アンアライインド データアクセス* P. A3-5 を参照して下さい。

a ビットは、0 としてエンコードする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];
    if wback then R[n] = R[n] + (if register_index then R[m] else 3*ebytes);
    D[d] = Replicate(MemU[address,ebytes], elements);
    D[d2] = Replicate(MemU[address+ebytes,ebytes], elements);
    D[d3] = Replicate(MemU[address+2*ebytes,ebytes], elements);
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.316 VLD4 (multiple 4-element structures)

This instruction loads multiple 4-element structures from memory into four registers, with de-interleaving. For more information, see *Element and structure load/store instructions* on page A4-27. Every element of each register is loaded. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VLD4<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD4<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15 14 13 12	11 10 9 8	7	6	5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 1	1 0 0 1	0	D	1 0	Rn	Vd	type	size	align	Rm

31 30 29 28	27 26 25 24	23	22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 1	0 1 0 0	0	D	1 0	Rn	Vd	type	size	align	Rm

```
if size == '11' then UNDEFINED;
case type of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d4 > 31 then UNPREDICTABLE;
```

Related encodings See *Advanced SIMD element or structure load/store instructions* on page A7-27

Assembler syntax

```
VLD4<c><q>.<size>  <list>, [<Rn>{@<align>}]          Rm = '1111'
VLD4<c><q>.<size>  <list>, [<Rn>{@<align>}]!         Rm = '1101'
VLD4<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>    Rm = other values
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VLD4` instruction must be unconditional.

`<size>` The data size. It must be one of:

- 8 encoded as size = 0b00
- 16 encoded as size = 0b01
- 32 encoded as size = 0b10.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.316 VLD4（複数の 4 エレメント構造のロード）

VLD4（複数の 4 エレメント構造のロード）は、複数の 4 エレメント構造をメモリから 4 つのレジスタに、インターリーブ解除を行ってロードします。詳細については、*エレメントおよび構造のロード / ストア命令* P. A4-27 を参照して下さい。各レジスタのすべてのエレメントがロードされます。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VLD4<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD4<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15 14 13 12	11 10 9 8	7	6	5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 1	1 0 0 1	0	D	1 0	Rn	Vd	type	size	align	Rm

31 30 29 28	27 26 25 24	23	22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1 1 1 1	0 1 0 0	0	D	1 0	Rn	Vd	type	size	align	Rm

```
if size == '11' then UNDEFINED;
case type of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d4 > 31 then UNPREDICTABLE;
```

関連エンコード *アドバンスト SIMD のエレメントまたは構造ロード / ストア命令* P. A7-27 を参照して下さい。

アセンブラ構文

```
VLD4<c><q>.<size>  <list>, [<Rn>{@<align>}]          Rm = '1111'
VLD4<c><q>.<size>  <list>, [<Rn>{@<align>}]!         Rm = '1101'
VLD4<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>    Rm = その他の値
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VLD4` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- 8 size = 0b00 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` The list of registers to load. It must be one of:

`{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>}`
encoded as D:Vd = `<Dd>`, type = 0b0000

`{<Dd>, <Dd+2>, <Dd+4>, <Dd+6>}`
encoded as D:Vd = `<Dd>`, type = 0b0001.

`<Rn>` Contains the base address for the access.

`<align>` The alignment. It can be one of:

- `64` 8-byte alignment, encoded as align = 0b01.
- `128` 16-byte alignment, encoded as align = 0b10.
- `256` 32-byte alignment, encoded as align = 0b11.
- **omitted** Standard alignment, see *Unaligned data access* on page A3-5. Encoded as align = 0b00.

`!` If present, specifies writeback.

`<Rm>` Contains an address offset applied after the access.

For more information about `<Rn>`, `!`, and `<Rm>`, see *Advanced SIMD addressing mode* on page A7-30.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 32);
    for e = 0 to elements-1
        Elem[D[d],e,esize] = MemU[address,ebytes];
        Elem[D[d2],e,esize] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e,esize] = MemU[address+2*ebytes,ebytes];
        Elem[D[d4],e,esize] = MemU[address+3*ebytes,ebytes];
        address = address + 4*ebytes;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

`<list>` ロードするレジスタのリスト。次のいずれかを使用できます。

`{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>}`
D:Vd = `<Dd>`、type = 0b0000 としてエンコードされます。

`{<Dd>, <Dd+2>, <Dd+4>, <Dd+6>}`
D:Vd = `<Dd>`、type = 0b0001 としてエンコードされます。

`<Rn>` アクセス用のベースアドレス。

`<align>` アライメント。次のいずれかを使用できます。

- `64` 8 バイトアライメント。align = 0b01 としてエンコードされます。
- `128` 16 バイトアライメント。align = 0b10 としてエンコードされます。
- `256` 32 バイトアライメント。align = 0b11 としてエンコードされます。
- **省略** 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。align = 0b00 としてエンコードされます。

`!` 存在する場合、ライトバックが指定されます。

`<Rm>` アクセス後に適用されるアドレスオフセット

`<Rn>`、`!`、および `<Rm>` の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 32);
    for e = 0 to elements-1
        Elem[D[d],e,esize] = MemU[address,ebytes];
        Elem[D[d2],e,esize] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e,esize] = MemU[address+2*ebytes,ebytes];
        Elem[D[d4],e,esize] = MemU[address+3*ebytes,ebytes];
        address = address + 4*ebytes;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.317 VLD4 (single 4-element structure to one lane)

This instruction loads one 4-element structure from memory into corresponding elements of four registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VLD4<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD4<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				size		1	1	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				size		1	1	index_align				Rm			

```
if size == '11' then SEE VLD4 (single 4-element structure to all lanes);
case size of
    when '00'
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '01'
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
    when '10'
        if index_align<1:0> == '11' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d4 > 31 then UNPREDICTABLE;
```

Assembler syntax

`VLD4<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VLD4<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VLD4<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VLD4` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.317 VLD4（1 つのレーンへの単一の 4 エレメント構造のロード）

VLD4（1 つのレーンへの単一の 4 エレメント構造のロード）は、1 つの 4 エレメント構造をメモリから、4 つのレジスタの対応するエレメントにロードします。レジスタの、ロード対象でないエレメントは変更されません。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VLD4<c>.<size> <list>, [<Rn>{@<align>}]{!}
VLD4<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				size		1	1	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				size		1	1	index_align				Rm			

```
if size == '11' then SEE VLD4 (single 4-element structure to all lanes);
case size of
    when '00'
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '01'
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
    when '10'
        if index_align<1:0> == '11' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d4 > 31 then UNPREDICTABLE;
```

アセンブラ構文

`VLD4<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VLD4<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VLD4<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VLD4` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<list> The registers containing the structure. Encoded with D:Vd = <Dd>. It must be one of:
{<Dd[x]>, <Dd+1[x]>, <Dd+2[x]>, <Dd+3[x]>} single-spaced registers, see Table A8-8.
{<Dd[x]>, <Dd+2[x]>, <Dd+4[x]>, <Dd+6[x]>} double-spaced registers, see Table A8-8. Not available if <size> == 8.

<Rn> The base address for the access.

<align> The alignment. It can be:
- 32 4-byte alignment, available only if <size> is 8.
- 64 8-byte alignment, available only if <size> is 16 or 32.
- 128 16-byte alignment, available only if <size> is 32.
- **omitted** Standard alignment, see *Unaligned data access* on page A3-5.

! If present, specifies writeback.

<Rm> Contains an address offset applied after the access.

For more information about <Rn>, !, and <Rm> see *Advanced SIMD addressing mode* on page A7-30.

Table A8-8 Encoding of index, alignment, and register spacing

	<size> == 8	<size> == 16	<size> == 32
Index	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
Single-spacing	-	index_align[1] = 0	index_align[2] = 0
Double-spacing	-	index_align[1] = 1	index_align[2] = 1
<align> omitted	index_align[0] = 0	index_align[0] = 0	index_align[1:0] = '00'
<align> == 32	index_align[0] = 1	-	-
<align> == 64	-	index_align[0] = 1	index_align[1:0] = '01'
<align> == 128	-	-	index_align[1:0] = '10'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 4*ebytes);
    Elem[D[d],index,esize] = MemU[address,ebytes];
    Elem[D[d2],index,esize] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index,esize] = MemU[address+2*ebytes,ebytes];
    Elem[D[d4],index,esize] = MemU[address+3*ebytes,ebytes];
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<list> 構造を含むレジスタ。D:Vd = <Dd> としてエンコードされます。次のいずれかを使用できます。
{<Dd[x]>, <Dd+1[x]>, <Dd+2[x]>, <Dd+3[x]>} シングルスペース レジスタ（表 A8-8 参照）。
{<Dd[x]>, <Dd+2[x]>, <Dd+4[x]>, <Dd+6[x]>} ダブルスペース レジスタ（表 A8-8 参照）。<size> ＝8 の場合には使用できません。

<Rn> アクセス用のベースアドレス。

<align> アライメント。次のいずれかを使用できます。
- 32 4 バイトアライメント。<size> が 8 の場合にのみ使用できます。
- 64 8 バイトアライメント。<size> が 16 または 32 の場合にのみ使用できます。
- 128 16 バイトアライメント。<size> が 32 の場合にのみ使用できます。
- **省略** 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。

! 存在する場合、ライトバックが指定されます。

<Rm> アクセス後に適用されるアドレスオフセット

<Rn>、!、および <Rm> の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

表 A8-8 インデクス、アライメント、およびレジスタスペーシングのエンコード

	<size> ＝8	<size> ＝16	<size> ＝32
インデクス	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
シングルスペーシング	–	index_align[1] = 0	index_align[2] = 0
ダブルスペーシング	–	index_align[1] = 1	index_align[2] = 1
<align> 省略	index_align[0] = 0	index_align[0] = 0	index_align[1:0] = '00'
<align> ＝32	index_align[0] = 1	–	–
<align> ＝64	–	index_align[0] = 1	index_align[1:0] = '01'
<align> ＝128	–	–	index_align[1:0] = '10'

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 4*ebytes);
    Elem[D[d],index,esize] = MemU[address,ebytes];
    Elem[D[d2],index,esize] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index,esize] = MemU[address+2*ebytes,ebytes];
    Elem[D[d4],index,esize] = MemU[address+3*ebytes,ebytes];
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.318 VLD4 (single 4-element structure to all lanes)

This instruction loads one 4-element structure from memory into all lanes of four registers. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VLD4<c>.<size> <list>, [<Rn>{ @<align>}]{!}
VLD4<c>.<size> <list>, [<Rn>{ @<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	1	1	1	size		T	a	Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	1	1	1	size		T	a	Rm			

```
if size == '11' && a == '0' then UNDEFINED;
if size == '11' then
    ebytes = 4;  elements = 2;  alignment = 16;
else
    ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
    if size == '10' then
        alignment = if a == '0' then 1 else 8;
    else
        alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d4 > 31 then UNPREDICTABLE;
```

Assembler syntax

```
VLD4<c><q>.<size>  <list>, [<Rn>{ @<align>}]          Rm = '1111'
VLD4<c><q>.<size>  <list>, [<Rn>{ @<align>}]!         Rm = '1101'
VLD4<c><q>.<size>  <list>, [<Rn>{ @<align>}], <Rm>    Rm = other values
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VLD4` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10 (or 0b11 for 16-byte alignment).

`<list>` The registers containing the structures. It must be one of:

`{<Dd[]>, <Dd+1[]>, <Dd+2[]>, <Dd+3[]>}`
single-spaced registers, encoded as D:Vd = `<Dd>`, T = 0

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.318 VLD4（全レーンへの単一の 4 エレメント構造のロード）

VLD4（全レーンへの単一の 4 エレメント構造のロード）は、1 つの 4 エレメント構造をメモリから 4 つのレジスタの全レーンにロードします。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VLD4<c>.<size> <list>, [<Rn>{ @<align>}]{!}
VLD4<c>.<size> <list>, [<Rn>{ @<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	1	1	1	size		T	a	Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	1	1	1	size		T	a	Rm			

```
if size == '11' && a == '0' then UNDEFINED;
if size == '11' then
    ebytes = 4;  elements = 2;  alignment = 16;
else
    ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
    if size == '10' then
        alignment = if a == '0' then 1 else 8;
    else
        alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d4 > 31 then UNPREDICTABLE;
```

アセンブラ構文

```
VLD4<c><q>.<size>  <list>, [<Rn>{ @<align>}]          Rm = '1111'
VLD4<c><q>.<size>  <list>, [<Rn>{ @<align>}]!         Rm = '1101'
VLD4<c><q>.<size>  <list>, [<Rn>{ @<align>}], <Rm>    Rm = その他の値
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VLD4` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10（または 16 バイトアライメントでは 0b11）としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

{<Dd[]>, <Dd+2[]>, <Dd+4[]>, <Dd+6[]>}
double-spaced register transfer, encoded as D:Vd = <Dd>, T = 1.

<Rn> The base address for the access.

<align> The alignment. It can be one of:

- 32 4-byte alignment, available only if <size> is 8, encoded as a = 1.
- 64 8-byte alignment, available only if <size> is 16 or 32, encoded as a = 1.
- 128 16-byte alignment, available only if <size> is 32, encoded as a = 1, size = 0b11.
- **omitted** Standard alignment, see *Unaligned data access* on page A3-5. Encoded as a = 0.

! If present, specifies writeback.

<Rm> Contains an address offset applied after the access.

For more information about <Rn>, !, and <Rm>, see *Advanced SIMD addressing mode* on page A7-30.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 4*ebytes);
    D[d] = Replicate(MemU[address,ebytes], elements);
    D[d2] = Replicate(MemU[address+ebytes,ebytes], elements);
    D[d3] = Replicate(MemU[address+2*ebytes,ebytes], elements);
    D[d4] = Replicate(MemU[address+3*ebytes,ebytes], elements);
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<list> 構造を含むレジスタ。次のいずれかを使用できます。

{<Dd[]>, <Dd+1[]>, <Dd+2[]>, <Dd+3[]>}
シングルスペース レジスタ。D:Vd = <Dd>、T = 0 としてエンコードされます。

{<Dd[]>, <Dd+2[]>, <Dd+4[]>, <Dd+6[]>}
ダブルスペース レジスタ転送。D:Vd = <Dd>、T = 1 としてエンコードされます。

<Rn> アクセス用のベースアドレス。

<align> アライメント。次のいずれかを使用できます。

- 32 4 バイトアライメント。<size> が 8 の場合にのみ使用できます。a = 1 としてエンコードされます。
- 64 8 バイトアライメント。<size> が 16 または 32 の場合にのみ使用できます。a = 1 としてエンコードされます。
- 128 16 バイトアライメント。<size> が 32 の場合にのみ使用できます。a = 1、size = 0b11 としてエンコードされます。
- **省略** 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。a = 0 としてエンコードされます。

! 存在する場合、ライトバックが指定されます。

<Rm> アクセス後に適用されるアドレスオフセット

<Rn>、!、および <Rm> の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 4*ebytes);
    D[d] = Replicate(MemU[address,ebytes], elements);
    D[d2] = Replicate(MemU[address+ebytes,ebytes], elements);
    D[d3] = Replicate(MemU[address+2*ebytes,ebytes], elements);
    D[d4] = Replicate(MemU[address+3*ebytes,ebytes], elements);
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.319 VLDM

Vector Load Multiple loads multiple extension registers from consecutive memory locations using an address from an ARM core register.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD

`VLDM{mode}<c> <Rn>{!}, <list>` `<list>` is consecutive 64-bit registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	1	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	1	imm8							

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '0' && U == '1' && W == '1' && Rn == '1101' then SEE VPOP;
if P == '1' && W == '0' then SEE VLDR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FLDMX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
```

Encoding T2 / A2 VFPv2, VFPv3

`VLDM{mode}<c> <Rn>{!}, <list>` `<list>` is consecutive 32-bit registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	0	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	0	imm8							

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '0' && U == '1' && W == '1' && Rn == '1101' then SEE VPOP;
if P == '1' && W == '0' then SEE VLDR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

Related encodings See *64-bit transfers between ARM core and extension registers* on page A7-32

FLDMX Encoding T1/A1 behaves as described by the pseudocode if imm8 is odd. However, there is no UAL syntax for such encodings and their use is deprecated. For more information, see *FLDMX, FSTMX* on page A8-101.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.319 VLDM

VLDM（ベクタ複数ロード）は、ARM コアレジスタからのアドレスを使用して、連続するメモリ位置から複数の拡張機能レジスタをロードします。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD

`VLDM{mode}<c> <Rn>{!}, <list>` `<list>` は連続する 64 ビットレジスタ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	1	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	1	imm8							

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '0' && U == '1' && W == '1' && Rn == '1101' then SEE VPOP;
if P == '1' && W == '0' then SEE VLDR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FLDMX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
```

エンコード T2/A2 VFPv2、VFPv3

`VLDM{mode}<c> <Rn>{!}, <list>` `<list>` は連続する 32 ビットレジスタ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	0	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	0	imm8							

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '0' && U == '1' && W == '1' && Rn == '1101' then SEE VPOP;
if P == '1' && W == '0' then SEE VLDR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

関連エンコード *ARM コアレジスタと拡張機能レジスタとの 64 ビット転送* P. A7-32 を参照して下さい。

FLDMX imm8 が奇数の場合、エンコード T1/A1 は擬似コードで記述されているように動作します。しかし、このようなエンコード用の UAL 構文は存在せず、このようなエンコードの使用は推奨されません。詳細については、*FLDMX*、*FSTMX* P. A8-101 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VLDM{<mode>}<c><q>{.<size>}  <Rn>{!}, <list>
```

where:

`<mode>` The addressing mode:

- `IA` Increment After. The consecutive addresses start at the address specified in `<Rn>`. This is the default and can be omitted. Encoded as P = 0, U = 1.
- `DB` Decrement Before. The consecutive addresses end just before the address specified in `<Rn>`. Encoded as P = 1, U = 0.

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<size>` An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers in `<list>`.

`<Rn>` The base register. The SP can be used. In the ARM instruction set, if `!` is not specified the PC can be used.

`!` Causes the instruction to write a modified value back to `<Rn>`. This is required if `<mode>` == `DB`, and is optional if `<mode>` == `IA`. Encoded as W = 1.

If `!` is omitted, the instruction does not change `<Rn>` in this way. Encoded as W = 0.

`<list>` The extension registers to be loaded, as a list of consecutively numbered doubleword (encoding T1 / A1) or singleword (encoding T2 / A2) registers, separated by commas and surrounded by brackets. It is encoded in the instruction by setting D and Vd to specify the first register in the list, and imm8 to twice the number of registers in the list (encoding T1 / A1) or the number of registers in the list (encoding T2 / A2). `<list>` must contain at least one register. If it contains doubleword registers it must not contain more than 16 registers.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);  NullCheckIfThumbEE(n);
    address = if add then R[n] else R[n]-imm32;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            S[d+r] = MemA[address,4];  address = address+4;
        else
            word1 = MemA[address,4];  word2 = MemA[address+4,4];  address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            D[d+r] = if BigEndian() then word1:word2 else word2:word1;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VLDM{<mode>}<c><q>{.<size>}  <Rn>{!}, <list>
```

各項目の説明については以下を参照して下さい。

`<mode>` アドレシングモード。

- `IA` ポストインクリメント。連続するアドレスは、`<Rn>` で指定されたアドレスから始まります。これはデフォルトであり、省略することができます。P = 0、U = 1 としてエンコードされます。
- `DB` プリデクリメント。連続するアドレスは、`<Rn>` で指定されたアドレスの 1 つ前で終わります。P = 1、U = 0 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<size>` オプションのデータサイズ指定子。指定する場合、`<list>` に含まれるレジスタのサイズ（32 ビットまたは 64 ビット）に等しいことが必要です。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。ARM 命令セットでは、`!` が指定されていない場合に PC を使用できます。

`!` 命令の実行によって変更された値を `<Rn>` に書き戻すように指示します。これは、`<mode>` == `DB` の場合に必須で、`<mode>` == `IA` の場合にはオプションです。W = 1 としてエンコードされます。

`!` を省略した場合は、命令によって `<Rn>` は変更されません。W = 0 としてエンコードされます。

`<list>` ロードされる拡張機能レジスタ。カンマで区切られ、角括弧で囲まれた、番号が連続しているダブルワード（エンコード T1/A1）レジスタまたはシングルワード（エンコード T2/A2）レジスタのリストとして示されます。リストの最初のレジスタを指定するために D および Vd を設定し、リストに含まれるレジスタの数の 2 倍（エンコード T1/A1）またはリストに含まれるレジスタの数（エンコード T2/A2）を指定するために imm8 を設定することで、命令にエンコードされます。`<list>` には、少なくとも 1 つのレジスタを含む必要があります。ダブルワードレジスタが含まれる場合、16 を超えるレジスタを含むことはできません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);  NullCheckIfThumbEE(n);
    address = if add then R[n] else R[n]-imm32;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            S[d+r] = MemA[address,4];  address = address+4;
        else
            word1 = MemA[address,4];  word2 = MemA[address+4,4];  address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            D[d+r] = if BigEndian() then word1:word2 else word2:word1;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.320 VLDR

This instruction loads a single extension register from memory, using an address from an ARM core register, with an optional offset.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD

```
VLDR<c> <Dd>, [<Rn>{, #+/-<imm>}]
VLDR<c> <Dd>, <label>
VLDR<c> <Dd>, [PC,#-0]                                  Special case
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	U	D	0	1	Rn				Vd				1	0	1	1	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	1	U	D	0	1	Rn				Vd				1	0	1	1	imm8							

```
single_reg = FALSE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(D:Vd);  n = UInt(Rn);
```

Encoding T2 / A2 VFPv2, VFPv3

```
VLDR<c> <Sd>, [<Rn>{, #+/-<imm>}]
VLDR<c> <Sd>, <label>
VLDR<c> <Sd>, [PC,#-0]                                  Special case
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	U	D	0	1	Rn				Vd				1	0	1	0	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	1	U	D	0	1	Rn				Vd				1	0	1	0	imm8							

```
single_reg = TRUE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(Vd:D);  n = UInt(Rn);
```

Assembler syntax

`VLDR<c><q>{.64} <Dd>, [<Rn> {, #+/-<imm>}]`	Encoding T1 / A1, immediate form
`VLDR<c><q>{.64} <Dd>, <label>`	Encoding T1 / A1, normal literal form
`VLDR<c><q>{.64} <Dd>, [PC, #+/-<imm>]`	Encoding T1 / A1, alternative literal form
`VLDR<c><q>{.32} <Sd>, [<Rn> {, #+/-<imm>}]`	Encoding T2 / A2, immediate form
`VLDR<c><q>{.32} <Sd>, <label>`	Encoding T2 / A2, normal literal form
`VLDR<c><q>{.32} <Sd>, [PC, #+/-<imm>]`	Encoding T2 / A2, alternative literal form

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`.32, .64` Optional data size specifiers.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.320 VLDR

VLDR は、ARM コアレジスタからのアドレスとオプションのオフセットを使用して、1 つの拡張機能レジスタをメモリからロードします。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD

```
VLDR<c> <Dd>, [<Rn>{, #+/-<imm>}]
VLDR<c> <Dd>, <label>
VLDR<c> <Dd>, [PC,#-0]                                  特殊なケース
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	U	D	0	1	Rn				Vd				1	0	1	1	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	1	U	D	0	1	Rn				Vd				1	0	1	1	imm8							

```
single_reg = FALSE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(D:Vd);  n = UInt(Rn);
```

エンコード T2/A2 VFPv2、VFPv3

```
VLDR<c> <Sd>, [<Rn>{, #+/-<imm>}]
VLDR<c> <Sd>, <label>
VLDR<c> <Sd>, [PC,#-0]                                  特殊なケース
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	U	D	0	1	Rn				Vd				1	0	1	0	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	1	U	D	0	1	Rn				Vd				1	0	1	0	imm8							

```
single_reg = TRUE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(Vd:D);  n = UInt(Rn);
```

アセンブラ構文

`VLDR<c><q>{.64} <Dd>, [<Rn> {, #+/-<imm>}]`	エンコード T1/A1、イミディエート形式
`VLDR<c><q>{.64} <Dd>, <label>`	エンコード T1/A1、通常のリテラル形式
`VLDR<c><q>{.64} <Dd>, [PC, #+/-<imm>]`	エンコード T1/A1、代替のリテラル形式
`VLDR<c><q>{.32} <Sd>, [<Rn> {, #+/-<imm>}]`	エンコード T2/A2、イミディエート形式
`VLDR<c><q>{.32} <Sd>, <label>`	エンコード T2/A2、通常のリテラル形式
`VLDR<c><q>{.32} <Sd>, [PC, #+/-<imm>]`	エンコード T2/A2、代替のリテラル形式

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`.32, .64` オプションのデータサイズ指定子。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<Dd> The destination register for a doubleword load.

<Sd> The destination register for a singleword load.

<Rn> The base register. The SP can be used.

+/- Is + or omitted if the immediate offset is to be added to the base register value (add == TRUE), or – if it is to be subtracted (add == FALSE). #0 and #-0 generate different instructions.

<imm> The immediate offset used to form the address. For the immediate forms of the syntax, <imm> can be omitted, in which case the #0 form of the instruction is assembled. Permitted values are multiples of 4 in the range 0 to 1020.

<label> The label of the literal data item to be loaded. The assembler calculates the required value of the offset from the Align(PC,4) value of this instruction to the label. Permitted values are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE.

For the literal forms of the instruction, the base register is encoded as '1111' to indicate that the PC is the base register.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);  NullCheckIfThumbEE(n);
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    if single_reg then
        S[d] = MemA[address,4];
    else
        word1 = MemA[address,4];  word2 = MemA[address+4,4];
        // Combine the word-aligned words in the correct order for current endianness.
        D[d] = if BigEndian() then word1:word2 else word2:word1;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<Dd> ダブルワードロードの場合のデスティネーションレジスタ

<Sd> シングルワードロードの場合のデスティネーションレジスタ

<Rn> ベースレジスタ。このレジスタには SP を指定できます。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、「+」にするか、省略します。減算する場合 (add == FALSE) は、「-」にします。#0 と #-0 では異なる命令が生成されます。

<imm> アドレスを形成するために使用されるイミディエートオフセット。イミディエート形式の構文では、<imm> を省略できます。この場合、#0 形式の命令がアセンブルされます。許容される値の範囲は、4 の倍数で 0 ～ 1020 です。

<label> ロードされるリテラルデータ項目のラベル。アセンブラは、この命令の Align(PC,4) の値からこのラベルまでのオフセットに必要な値を計算します。許容される値の範囲は、4 の倍数で - 1020 ～ 1020 です。

オフセットが 0 または正の場合、imm32 はオフセットと等しく、add == TRUE になります。

オフセットが負の場合、imm32 は負のオフセットと等しく、add == FALSE になります。

リテラル形式の命令では、PC がベースレジスタであることを示すため、ベースレジスタが '1111' としてエンコードされます。

代替構文では、通常の構文では指定できない 0 の減算など、オフセットおよびイミディエートオフセットの加算または減算を個別に指定することが許可されます。詳細については、*UAL 命令構文でのラベルの使用* P. A4-5 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);  NullCheckIfThumbEE(n);
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    if single_reg then
        S[d] = MemA[address,4];
    else
        word1 = MemA[address,4];  word2 = MemA[address+4,4];
        // Combine the word-aligned words in the correct order for current endianness.
        D[d] = if BigEndian() then word1:word2 else word2:word1;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.321 VMAX, VMIN (integer)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Encoding T1 / A1 Advanced SIMD

```
V<op><c>.<dt> <Qd>, <Qn>, <Qm>
V<op><c>.<dt> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	1	0	N	Q	M	op	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	1	0	N	Q	M	op	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.321 VMAX、VMIN（整数）

VMAX（ベクタ最大（整数））は、2 つのベクタの対応するエレメントを比較し、各ペアの大きい方をデスティネーションベクタの対応するエレメントにコピーします。

VMIN（ベクタ最小（整数））は、2 つのベクタの対応するエレメントを比較し、各ペアの小さい方をデスティネーションベクタの対応するエレメントにコピーします。

オペランドベクタのエレメントには、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットの符号付き整数
- 8 ビット、16 ビット、または 32 ビットの符号なし整数

結果ベクタのエレメントのサイズは、オペランドベクタのエレメントと同じです。

エンコード T1/A1 アドバンスト SIMD

```
V<op><c>.<dt> <Qd>, <Qn>, <Qm>
V<op><c>.<dt> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	1	0	N	Q	M	op	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	1	0	N	Q	M	op	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`V<op><c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`V<op><c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<op>` Must be one of:
- `MAX` encoded as op = 0
- `MIN` encoded as op = 1.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VMAX` or `VMIN` instruction must be unconditional.

`<dt>` The data types for the elements of the vectors. It must be one of:
- `S8` size = 0b00, U = 0
- `S16` size = 0b01, U = 0
- `S32` size = 0b10, U = 0
- `U8` size = 0b00, U = 1
- `U16` size = 0b01, U = 1
- `U32` size = 0b10, U = 1.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if maximum then Max(op1,op2) else Min(op1,op2);
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`V<op><c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`V<op><c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<op>` 次のいずれかを使用できます。
- `MAX` op = 0 としてエンコードされます。
- `MIN` op = 1 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VMAX` および `VMIN` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。
- `S8` size = 0b00、U = 0
- `S16` size = 0b01、U = 0
- `S32` size = 0b10、U = 0
- `U8` size = 0b00、U = 1
- `U16` size = 0b01、U = 1
- `U32` size = 0b10、U = 1

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if maximum then Max(op1,op2) else Min(op1,op2);
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.322 VMAX, VMIN (floating-point)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements are 32-bit floating-point numbers.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
V<op><c>.F32 <Qd>, <Qn>, <Qm>
V<op><c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	op	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	op	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
maximum = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.322 VMAX、VMIN（浮動小数点数）

VMAX（ベクタ最大（浮動小数点数））は、2 つのベクタの対応するエレメントを比較し、各ペアの大きい方をデスティネーションベクタの対応するエレメントにコピーします。

VMIN（ベクタ最小（浮動小数点数））は、2 つのベクタの対応するエレメントを比較し、各ペアの小さい方をデスティネーションベクタの対応するエレメントにコピーします。

オペランドベクタのエレメントは、32 ビットの浮動小数点数です。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
V<op><c>.F32 <Qd>, <Qn>, <Qm>
V<op><c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	op	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	op	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
maximum = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`V<op><c><q>.F32 {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`V<op><c><q>.F32 {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<op>` Must be one of:
- `MAX` encoded as op = 0
- `MIN` encoded as op = 1.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VMAX` or `VMIN` instruction must be unconditional.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            Elem[D[d+r],e,esize] = if maximum then FPMax(op1,op2,FALSE) else FPMin(op1,op2,FALSE);
```

Exceptions

Undefined Instruction.

Floating-point maximum and minimum

- max(+0.0, –0.0) = +0.0
- min(+0.0, –0.0) = –0.0
- If any input is a NaN, the corresponding result element is the default NaN.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`V<op><c><q>.F32 {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`V<op><c><q>.F32 {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<op>` 次のいずれかを使用できます。
- `MAX` op = 0 としてエンコードされます。
- `MIN` op = 1 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VMAX` および `VMIN` は常に無条件命令です。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            Elem[D[d+r],e,esize] = if maximum then FPMax(op1,op2,FALSE) else FPMin(op1,op2,FALSE);
```

例外

未定義命令

浮動小数点数の最大と最小

- max(+0.0, -0.0) = +0.0
- min(+0.0, -0.0) = -0.0
- いずれかの入力が NaN の場合、対応する結果エレメントはデフォルト NaN になります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.323 VMLA, VMLAL, VMLS, VMLSL (integer)

Vector Multiply Accumulate and Vector Multiply Subtract multiply corresponding elements in two vectors, and either add the products to, or subtract them from, the corresponding elements of the destination vector. Vector Multiply Accumulate Long and Vector Multiply Subtract Long do the same thing, but with destination vector elements that are twice as long as the elements that are multiplied.

Encoding T1 / A1 Advanced SIMD

```
V<op><c>.<dt> <Qd>, <Qn>, <Qm>
V<op><c>.<dt> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	op	1	1	1	1	0	D	size		Vn				Vd				1	0	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	op	0	D	size		Vn				Vd				1	0	0	1	N	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 Advanced SIMD

```
V<op>L<c>.<dt> <Qd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				1	0	op	0	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				1	0	op	0	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.323 VMLA、VMLAL、VMLS、VMLSL（整数）

VMLA、VMLS（ベクタ積和演算、ベクタ積減算（整数））は、2 つのベクタの対応するエレメントを乗算し、その積を、デスティネーションベクタの対応するエレメントに加算するか、デスティネーションベクタの対応するエレメントから減算します。VMLAL、VMLSL（ベクタロング積和演算、ベクタロング積減算（整数））は、これと同じ処理を実行しますが、デスティネーションベクタのエレメントの長さが、乗算されるエレメントの 2 倍になります。

エンコード T1/A1 アドバンスト SIMD

```
V<op><c>.<dt> <Qd>, <Qn>, <Qm>
V<op><c>.<dt> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	op	1	1	1	1	0	D	size		Vn				Vd				1	0	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	op	0	D	size		Vn				Vd				1	0	0	1	N	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 アドバンスト SIMD

```
V<op>L<c>.<dt> <Qd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				1	0	op	0	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				1	0	op	0	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`V<op><c><q>.<type><size>  <Qd>, <Qn>, <Qm>`	Encoding T1 / A1, Q = 1
`V<op><c><q>.<type><size>  <Dd>, <Dn>, <Dm>`	Encoding T1 / A1, Q = 0
`V<op>L<c><q>.<type><size>  <Qd>, <Dn>, <Dm>`	Encoding T2 / A2

where:

`<op>` Must be either `MLA` (op = 0) or `MLS` (op = 1).

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VMLA`, `VMLAL`, `VMLS`, or `VMLSL` instruction must be unconditional.

`<type>` The data type for the elements of the operands. It must be one of:

- `S` Optional in encoding T1 / A1. U = 0 in encoding T2 / A2.
- `U` Optional in encoding T1 / A1. U = 1 in encoding T2 / A2.
- `I` Available only in encoding T1 / A1.

`<size>` The data size for the elements of the operands. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

`<Qd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a long operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[D[n+r],e,esize],unsigned) * Int(Elem[D[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Q[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + addend;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`V<op><c><q>.<type><size>  <Qd>, <Qn>, <Qm>`	エンコード T1/A1、Q = 1
`V<op><c><q>.<type><size>  <Dd>, <Dn>, <Dm>`	エンコード T1/A1、Q = 0
`V<op>L<c><q>.<type><size>  <Qd>, <Dn>, <Dm>`	エンコード T2/A2

各項目の説明については以下を参照して下さい。

`<op>` `MLA` (op = 0) または `MLS` (op = 1) のいずれかを使用できます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VMLA`、`VMLAL`、`VMLS`、`VMLSL` は常に無条件命令です。

`<type>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S` エンコード T1/A1 ではオプションです。エンコード T2/A2 では U = 0。
- `U` エンコード T1/A1 ではオプションです。エンコード T2/A2 では U = 1。
- `I` エンコード T1/A1 でのみ使用できます。

`<size>` オペランドのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Qd>, <Dn>, <Dm>` ロング演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[D[n+r],e,esize],unsigned) * Int(Elem[D[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Q[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + addend;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.324 VMLA, VMLS (floating-point)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

Vector Multiply Subtract multiplies corresponding elements in two vectors, subtracts the products from corresponding elements of the destination vector, and places the results in the destination vector.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
V<op><c>.F32 <Qd>, <Qn>, <Qm>
V<op><c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	op	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	op	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  add = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
V<op><c>.F64 <Dd>, <Dn>, <Dm>
V<op><c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	0	0	Vn				Vd				1	0	1	sz	N	op	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	D	0	0	Vn				Vd				1	0	1	sz	N	op	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
advsimd = FALSE;  dp_operation = (sz == '1');  add = (op == '0');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP vectors Encoding T2 / A2 can operate on VFP vectors under control of the FPSCR.LEN and FPSCR.STRIDE bits. For details see Appendix F *VFP Vector Operation Support*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.324 VMLA、VMLS（浮動小数点数）

VMLA（ベクタ積和演算（浮動小数点数））は、2 つのベクタの対応するエレメントを乗算し、結果をデスティネーションベクタのエレメントに加算します。

VMLS（ベクタ積減算（浮動小数点数））は、2 つのベクタの対応するエレメントを乗算し、その積をデスティネーションベクタの対応するエレメントから減算し、結果をデスティネーションベクタに配置します。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
V<op><c>.F32 <Qd>, <Qn>, <Qm>
V<op><c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	op	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	op	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  add = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
V<op><c>.F64 <Dd>, <Dn>, <Dm>
V<op><c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	0	0	Vn				Vd				1	0	1	sz	N	op	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	D	0	0	Vn				Vd				1	0	1	sz	N	op	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
advsimd = FALSE;  dp_operation = (sz == '1');  add = (op == '0');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP ベクタ エンコード T2/A2 は、FPSCR.LEN ビットと FPSCR.STRIDE ビットの制御の下で VFP ベクタに作用することができます。詳細については、付録 F *「VFP ベクタ演算のサポート」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`V<op><c><q>.F32 <Qd>, <Qn>, <Qm>`	Encoding T1 / A1, Q = 1, sz = 0
`V<op><c><q>.F32 <Dd>, <Dn>, <Dm>`	Encoding T1 / A1, Q = 0, sz = 0
`V<op><c><q>.F64 <Dd>, <Dn>, <Dm>`	Encoding T2 / A2, sz = 1
`V<op><c><q>.F32 <Sd>, <Sn>, <Sm>`	Encoding T2 / A2, sz = 0

where:

`<op>`	Must be either `MLA` (op = 0) or `MLS` (op = 1).
`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VMLA` or `VMLS` instruction must be unconditional.
`<Qd>, <Qn>, <Qm>`	The destination vector and the operand vectors, for a quadword operation.
`<Dd>, <Dn>, <Dm>`	The destination vector and the operand vectors, for a doubleword operation.
`<Sd>, <Sn>, <Sm>`	The destination vector and the operand vectors, for a singleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], FALSE);
                addend = if add then product else FPNeg(product);
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], addend, FALSE);
    else             // VFP instruction
        if dp_operation then
            addend = if add then FPMul(D[n], D[m], TRUE) else FPNeg(FPMul(D[n], D[m], TRUE));
            D[d] = FPAdd(D[d], addend, TRUE);
        else
            addend = if add then FPMul(S[n], S[m], TRUE) else FPNeg(FPMul(S[n], S[m], TRUE));
            S[d] = FPAdd(S[d], addend, TRUE);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, Overflow, Underflow, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`V<op><c><q>.F32 <Qd>, <Qn>, <Qm>`	エンコード T1/A1、Q = 1、sz = 0
`V<op><c><q>.F32 <Dd>, <Dn>, <Dm>`	エンコード T1/A1、Q = 0、sz = 0
`V<op><c><q>.F64 <Dd>, <Dn>, <Dm>`	エンコード T2/A2、sz = 1
`V<op><c><q>.F32 <Sd>, <Sn>, <Sm>`	エンコード T2/A2、sz = 0

各項目の説明については以下を参照して下さい。

`<op>`	`MLA` (op = 0) または `MLS` (op = 1) のいずれかを使用できます。
`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VMLA` および `VMLS` は常に無条件命令です。
`<Qd>, <Qn>, <Qm>`	クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Dd>, <Dn>, <Dm>`	ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Sd>, <Sn>, <Sm>`	シングルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], FALSE);
                addend = if add then product else FPNeg(product);
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], addend, FALSE);
    else             // VFP instruction
        if dp_operation then
            addend = if add then FPMul(D[n], D[m], TRUE) else FPNeg(FPMul(D[n], D[m], TRUE));
            D[d] = FPAdd(D[d], addend, TRUE);
        else
            addend = if add then FPMul(S[n], S[m], TRUE) else FPNeg(FPMul(S[n], S[m], TRUE));
            S[d] = FPAdd(S[d], addend, TRUE);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、オーバフロー、アンダーフロー、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.325 VMLA, VMLAL, VMLS, VMLSL (by scalar)

Vector Multiply Accumulate and Vector Multiply Subtract multiply elements of a vector by a scalar, and either add the products to, or subtract them from, corresponding elements of the destination vector. Vector Multiply Accumulate Long and Vector Multiply Subtract Long do the same thing, but with destination vector elements that are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars* on page A7-9.

Encoding T1 / A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

```
V<op><c>.<dt> <Qd>, <Qn>, <Dm[x]>
V<op><c>.<dt> <Dd>, <Dn>, <Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	size		Vn				Vd				0	op	0	F	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	size		Vn				Vd				0	op	0	F	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Encoding T2 / A2 Advanced SIMD

```
V<op>L<c>.<dt> <Qd>, <Dn>, <Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				0	op	1	0	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				0	op	1	0	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.325 VMLA、VMLAL、VMLS、VMLSL（スカラ）

VMLA、VMLS（ベクタ積和演算、ベクタ積減算（スカラ））は、ベクタのエレメントとスカラを乗算し、その積を、デスティネーションベクタの対応するエレメントに加算するか、デスティネーションベクタの対応するエレメントから減算します。VMLAL、VMLSL（ベクタロング積和演算、ベクタロング積減算（スカラ））は、これと同じ処理を実行しますが、デスティネーションベクタのエレメントの長さが、乗算されるエレメントの 2 倍の長さになります。

スカラの詳細については、*アドバンスト SIMD スカラ* P. A7-9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは F = 1 は未定義）

```
V<op><c>.<dt> <Qd>, <Qn>, <Dm[x]>
V<op><c>.<dt> <Dd>, <Dn>, <Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	size		Vn				Vd				0	op	0	F	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	size		Vn				Vd				0	op	0	F	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

エンコード T2/A2 アドバンスト SIMD

```
V<op>L<c>.<dt> <Qd>, <Dn>, <Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				0	op	1	0	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				0	op	1	0	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`V<op><c><q>.<type><size>  <Qd>, <Qn>, <Dm[x]>`	Encoding T1 / A1, Q = 1
`V<op><c><q>.<type><size>  <Dd>, <Dn>, <Dm[x]>`	Encoding T1 / A1, Q = 0
`V<op>L<c><q>.<type><size>  <Qd>, <Dn>, <Dm[x]>`	Encoding T2 / A2

where:

`<op>` Must be either `MLA` (encoded as op = 0) or `MLS` (encoded as op = 1).

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VMLA`, `VMLAL`, `VMLS`, or `VMLSL` instruction must be unconditional.

`<type>` The data type for the elements of the operands. It must be one of:

- `S` encoding T2 / A2, U = '0'.
- `U` encoding T2 / A2, U = '1'.
- `I` encoding T1 / A1, F = '0'.
- `F` encoding T1 / A1, F = '1'. `<size>` must be 32.

`<size>` The operand element data size. It can be 16 (size = '01') or 32 (size = '10').

`<Qd>, <Qn>` The accumulate vector, and the operand vector, for a quadword operation.

`<Dd>, <Dn>` The accumulate vector, and the operand vector, for a doubleword operation.

`<Qd>, <Dn>` The accumulate vector, and the operand vector, for a long operation.

`<Dm[x]>` The scalar. `Dm` is restricted to D0-D7 if `<size>` is 16, or D0-D15 otherwise.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[D[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,FALSE) else FPNeg(FPMul(op1,op2,FALSE));
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], fp_addend, FALSE);
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Q[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + addend;
```

Exceptions

Undefined Instruction. Floating-point exceptions: Input Denormal, Invalid Operation, Overflow, Underflow, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`V<op><c><q>.<type><size>  <Qd>, <Qn>, <Dm[x]>`	エンコード T1/A1、Q = 1
`V<op><c><q>.<type><size>  <Dd>, <Dn>, <Dm[x]>`	エンコード T1/A1、Q = 0
`V<op>L<c><q>.<type><size>  <Qd>, <Dn>, <Dm[x]>`	エンコード T2/A2

各項目の説明については以下を参照して下さい。

`<op>` `MLA`（op = 0 としてエンコード）または `MLS`（op = 1 としてエンコード）のいずれかを使用できます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VMLA`、`VMLAL`、`VMLS`、`VMLSL` は常に無条件命令です。

`<type>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S` エンコード T2/A2、U = '0'
- `U` エンコード T2/A2、U = '1'
- `I` エンコード T1/A1、F = '0'
- `F` エンコード T1/A1、F = '1'（`<size>` は常に 32）

`<size>` オペランドのエレメントのデータサイズ。16(size = '01') または 32(size = '10') を指定できます。

`<Qd>, <Qn>` クワッドワード演算の場合の累積ベクタとオペランドベクタ

`<Dd>, <Dn>` ダブルワードワード演算の場合の累積ベクタとオペランドベクタ

`<Qd>, <Dn>` ロング演算の場合の累積ベクタとオペランドベクタ

`<Dm[x]>` スカラ。`<size>` が 16 の場合は `Dm` は D0 ～ D7 に限定され、それ以外の場合は D0 ～ D15 に限定されます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[D[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,FALSE) else FPNeg(FPMul(op1,op2,FALSE));
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], fp_addend, FALSE);
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Q[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + addend;
```

例外

未定義命令浮動小数点例外：入力非正規化、無効演算、オーバフロー、アンダーフロー、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.326 VMOV (immediate)

This instruction places an immediate constant into every element of the destination register.

Encoding T1 / A1 Advanced SIMD

```
VMOV<c>.<dt> <Qd>, #<imm>
VMOV<c>.<dt> <Dd>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd				cmode				0	Q	op	1	imm4			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd				cmode				0	Q	op	1	imm4			

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE VORR (immediate);
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VMOV<c>.F64 <Dd>, #<imm>
VMOV<c>.F32 <Sd>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	imm4H				Vd				1	0	1	sz	(0)	0	(0)	0	imm4L			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	imm4H				Vd				1	0	1	sz	(0)	0	(0)	0	imm4L			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
single_register = (sz == '0');  advsimd = FALSE;
if single_register then
    d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L, 32);
else
    d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L, 64);  regs = 1;
```

Related encodings See *One register and a modified immediate value* on page A7-21

VFP vectors Encoding T2 / A2 can operate on VFP vectors under control of the FPSCR.LEN and FPSCR.STRIDE bits. For details see Appendix F *VFP Vector Operation Support*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.326 VMOV（イミディエート）

VMOV（イミディエート）は、イミディエート定数をデスティネーションレジスタのすべてのエレメントに配置します。

エンコード T1/A1 アドバンスト SIMD

```
VMOV<c>.<dt> <Qd>, #<imm>
VMOV<c>.<dt> <Dd>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd				cmode				0	Q	op	1	imm4			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd				cmode				0	Q	op	1	imm4			

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE VORR (immediate);
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VMOV<c>.F64 <Dd>, #<imm>
VMOV<c>.F32 <Sd>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	imm4H				Vd				1	0	1	sz	(0)	0	(0)	0	imm4L			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	imm4H				Vd				1	0	1	sz	(0)	0	(0)	0	imm4L			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
single_register = (sz == '0');  advsimd = FALSE;
if single_register then
    d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L, 32);
else
    d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L, 64);  regs = 1;
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

VFP ベクタ エンコード T2/A2 は、FPSCR.LEN ビットと FPSCR.STRIDE ビットの制御の下で VFP ベクタに作用することができます。詳細については、付録 F *「VFP ベクタ演算のサポート」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VMOV<c><q>.<dt>  <Qd>, #<imm>`	Encoding T1 / A1, Q = 1
`VMOV<c><q>.<dt>  <Dd>, #<imm>`	Encoding T1 / A1, Q = 0
`VMOV<c>.F64  <Dd>, #<imm>`	Encoding T2 / A2, sz = 1
`VMOV<c>.F32  <Sd>, #<imm>`	Encoding T2 / A2, sz = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VMOV` (immediate) instruction must be unconditional.

`<dt>` The data type. It must be one of `I8`, `I16`, `I32`, `I64`, or `F32`.

`<Qd>` The destination register for a quadword operation.

`<Dd>` The destination register for a doubleword operation.

`<Sd>` The destination register for a singleword operation.

`<imm>` A constant of the type specified by `<dt>`. This constant is replicated enough times to fill the destination register. For example, `VMOV.I32 D0,#10` writes `0x0000000A0000000A` to D0.

For the range of constants available, and the encoding of `<dt>` and `<imm>`, see:

- *One register and a modified immediate value* on page A7-21 for encoding T1 / A1
- *VFP data-processing instructions* on page A7-24 for encoding T2 / A2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S[d] = imm32;
    else
        for r = 0 to regs-1
            D[d+r] = imm64;
```

Exceptions

Undefined Instruction.

Pseudo-instructions

One register and a modified immediate value on page A7-21 describes pseudo-instructions with a combination of `<dt>` and `<imm>` that is not supported by hardware, but that generates the same destination register value as a different combination that is supported by hardware.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VMOV<c><q>.<dt>  <Qd>, #<imm>`	エンコード T1/A1、Q = 1
`VMOV<c><q>.<dt>  <Dd>, #<imm>`	エンコード T1/A1、Q = 0
`VMOV<c>.F64  <Dd>, #<imm>`	エンコード T2/A2、sz = 1
`VMOV<c>.F32  <Sd>, #<imm>`	エンコード T2/A2、sz = 0

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VMOV`（イミディエート）は常に無条件命令です。

`<dt>` データ型。`I8`、`I16`、`I32`、`I64`、`F32` のいずれかを使用できます。

`<Qd>` クワッドワード演算の場合のデスティネーションレジスタ

`<Dd>` ダブルワード演算の場合のデスティネーションレジスタ

`<Sd>` シングルワード演算の場合のデスティネーションレジスタ

`<imm>` `<dt>` で指定されたデータ型の定数。この定数は、デスティネーションレジスタを満たすのに必要な回数だけ複製されます。例えば、`VMOV.I32 D0,#10` では、`0x0000000A0000000A` が D0 に書き込まれます。

使用可能な定数の範囲と、`<dt>` および `<imm>` のエンコードの詳細については、次のセクションを参照して下さい。

- *1 つのレジスタと修飾イミディエート値* P. A7-21（エンコード T1/A1）
- *VFP のデータ処理命令* P. A7-24（エンコード T2/A2）

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S[d] = imm32;
    else
        for r = 0 to regs-1
            D[d+r] = imm64;
```

例外

未定義命令

擬似命令

ハードウェアでサポートされていないにもかかわらず、ハードウェアでサポートされている別の組み合わせと同じデスティネーションレジスタの値を生成する、`<dt>` と `<imm>` の組み合わせを持つ擬似命令については、*1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.327 VMOV (register)

This instruction copies the contents of one register to another.

Encoding T1 / A1 Advanced SIMD

```
VMOV<c> <Qd>, <Qm>
VMOV<c> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	0	Vm				Vd				0	0	0	1	M	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	0	Vm				Vd				0	0	0	1	M	Q	M	1	Vm			

```
if !Consistent(M) || !Consistent(Vm) then SEE VORR (register);
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VMOV<c>.F64 <Dd>, <Dm>
VMOV<c>.F32 <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	1	sz	0	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	1	sz	0	1	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
single_register = (sz == '0');  advsimd = FALSE;
if single_register then
    d = UInt(Vd:D);  m = UInt(Vm:M);
else
    d = UInt(D:Vd);  m = UInt(M:Vm);  regs = 1;
```

VFP vectors Encoding T2 / A2 can operate on VFP vectors under control of the FPSCR.LEN and FPSCR.STRIDE bits. For details see Appendix F *VFP Vector Operation Support*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.327 VMOV（レジスタ）

VMOV（レジスタ）は、レジスタの内容を他のレジスタにコピーします。

エンコード T1/A1 アドバンスト SIMD

```
VMOV<c> <Qd>, <Qm>
VMOV<c> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	0	Vm				Vd				0	0	0	1	M	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	0	Vm				Vd				0	0	0	1	M	Q	M	1	Vm			

```
if !Consistent(M) || !Consistent(Vm) then SEE VORR (register);
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VMOV<c>.F64 <Dd>, <Dm>
VMOV<c>.F32 <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	1	sz	0	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	1	sz	0	1	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
single_register = (sz == '0');  advsimd = FALSE;
if single_register then
    d = UInt(Vd:D);  m = UInt(Vm:M);
else
    d = UInt(D:Vd);  m = UInt(M:Vm);  regs = 1;
```

VFP ベクタ エンコード T2/A2 は、FPSCR.LEN ビットと FPSCR.STRIDE ビットの制御の下で VFP ベクタに作用することができます。詳細については、付録 F「*VFP ベクタ演算のサポート*」を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VMOV<c><q>{.<dt>}  <Qd>, <Qm>`	Encoding T1 / A1, Q = 1
`VMOV<c><q>{.<dt>}  <Dd>, <Dm>`	Encoding T1 / A1, Q = 0
`VMOV<c><q>.F64  <Dd>, <Dm>`	Encoding T2 / A2, sz = 1
`VMOV<c><q>.F32  <Sd>, <Sm>`	Encoding T2 / A2, sz = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VMOV` (register) instruction must be unconditional.

`<dt>` An optional data type. `<dt>` must not be `F64`, but it is otherwise ignored.

`<Qd>, <Qm>` The destination register and the source register, for a quadword operation.

`<Dd>, <Dm>` The destination register and the source register, for a doubleword operation.

`<Sd>, <Sm>` The destination register and the source register, for a singleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S[d] = S[m];
    else
        for r = 0 to regs-1
            D[d+r] = D[m+r];
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VMOV<c><q>{.<dt>}  <Qd>, <Qm>`	エンコード T1/A1、Q = 1
`VMOV<c><q>{.<dt>}  <Dd>, <Dm>`	エンコード T1/A1、Q = 0
`VMOV<c><q>.F64  <Dd>, <Dm>`	エンコード T2/A2、sz = 1
`VMOV<c><q>.F32  <Sd>, <Sm>`	エンコード T2/A2、sz = 0

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VMOV`（レジスタ）は常に無条件命令です。

`<dt>` オプションのデータ型。`<dt>` として `F64` を指定することはできませんが、その他の場合は無視されます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションレジスタとソースレジスタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションレジスタとソースレジスタ

`<Sd>, <Sm>` シングルワード演算の場合のデスティネーションレジスタとソースレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S[d] = S[m];
    else
        for r = 0 to regs-1
            D[d+r] = D[m+r];
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.328 VMOV (ARM core register to scalar)

This instruction copies a byte, halfword, or word from an ARM core register into an Advanced SIMD scalar.

On a VFP-only system, this instruction transfers one word to the upper or lower half of a double-precision floating-point register from an ARM core register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see *Advanced SIMD scalars* on page A7-9.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD if `opc1 == '0x' && opc2 == '00'`
Advanced SIMD otherwise

```
VMOV<c>.<size> <Dd[x]>, <Rt>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	opc1		0	Vd				Rt				1	0	1	1	D	opc2		1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	opc1		0	Vd				Rt				1	0	1	1	D	opc2		1	(0)	(0)	(0)	(0)

```
case opc1:opc2 of
    when '1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when '0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when '0x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when '0x10'  UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);
if t == 15 || (CurrentInstrSet() != InstrSet_ARM && t == 13) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.328 VMOV（ARM コアレジスタからスカラへのコピー）

VMOV（ARM コアレジスタからスカラへのコピー）は、バイト、ハーフワード、またはワードを、ARM コアレジスタからアドバンスト SIMD スカラにコピーします。

VFP のみのシステムでは、この命令は 1 ワードを ARM コアレジスタから、倍精度浮動小数点レジスタの上位半分または下位半分に転送します。この動作は、アドバンスト SIMD シングルワード転送と同じです。

スカラの詳細については、*アドバンスト SIMD スカラ* P. A7-9 を参照して下さい。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD（`opc1 == '0x' && opc2 == '00'` の場合）
アドバンスト SIMD（その他の場合）

```
VMOV<c>.<size> <Dd[x]>, <Rt>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	opc1		0	Vd				Rt				1	0	1	1	D	opc2		1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	opc1		0	Vd				Rt				1	0	1	1	D	opc2		1	(0)	(0)	(0)	(0)

```
case opc1:opc2 of
    when '1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when '0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when '0x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when '0x10'  UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);
if t == 15 || (CurrentInstrSet() != InstrSet_ARM && t == 13) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VMOV<c>{.<size>}  <Dd[x]>, <Rt>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<size> The data size. It must be one of:

- 8 Encoded as opc1<1> = 1. [x] is encoded in opc1<0>, opc2.
- 16 Encoded as opc1<1>, opc2<0> = 0b01. [x] is encoded in opc1<0>, opc2<1>.
- 32 Encoded as opc1<1>, opc2 = 0b000. [x] is encoded in opc1<0>.
- **omitted** equivalent to 32.

<Dd[x]> The scalar. The register <Dd> is encoded in D:Vd. For details of how [x] is encoded, see the description of <size>.

<Rt> The source ARM core register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    Elem[D[d+r],index,esize] = R[t]<esize-1:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VMOV<c>{.<size>}  <Dd[x]>, <Rt>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<size> データサイズ。次のいずれかを使用できます。

- 8 opc1〈1〉= 1 としてエンコードされます。[x] は、opc1〈0〉、opc2 にエンコードされます。
- 16 opc1〈1〉、opc2〈0〉= 0b01 としてエンコードされます。[x] は、opc1〈0〉、opc2〈1〉にエンコードされます。
- 32 opc1〈1〉、opc2 = 0b000 としてエンコードされます。[x] は、opc1〈0〉にエンコードされます。
- **省略** 32 と等価です。

<Dd[x]> スカラ。レジスタ <Dd> は、D:Vd にエンコードされます。[x] のエンコード方法の詳細については、<size> の説明を参照して下さい。

<Rt> ソース ARM コアレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    Elem[D[d+r],index,esize] = R[t]<esize-1:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.329 VMOV (scalar to ARM core register)

This instruction copies a byte, halfword, or word from an Advanced SIMD scalar to an ARM core register. Bytes and halfwords can be either zero-extended or sign-extended.

On a VFP-only system, this instruction transfers one word from the upper or lower half of a double-precision floating-point register to an ARM core register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see *Advanced SIMD scalars* on page A7-9.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD if `opc1 == '0x' && opc2 == '00'`
Advanced SIMD otherwise

```
VMOV<c>.<dt> <Rt>, <Dn[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	U	opc1		1	Vn				Rt				1	0	1	1	N	opc2		1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	U	opc1		1	Vn				Rt				1	0	1	1	N	opc2		1	(0)	(0)	(0)	(0)

```
case U:opc1:opc2 of
    when ‘x1xxx’  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when ‘x0xx1’  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when ‘00x00’  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when ‘10x00’  UNDEFINED;
    when ‘x0x10’  UNDEFINED;
t = UInt(Rt);  n = UInt(N:Vn);  unsigned = (U == ‘1’);
if t == 15 || (CurrentInstrSet() != InstrSet_ARM && t == 13) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.329 VMOV（スカラから ARM コアレジスタへのコピー）

この命令は、バイト、ハーフワード、またはワードを、アドバンスト SIMD スカラから ARM コアレジスタにコピーします。バイトおよびハーフワードをゼロ拡張または符号拡張することができます。

VFP のみのシステムでは、この命令は 1 ワードを、倍精度浮動小数点レジスタの上位半分または下位半分から ARM コアレジスタに転送します。この動作は、アドバンスト SIMD シングルワード転送と同じです。

スカラの詳細については、*アドバンスト SIMD スカラ* P. A7-9 を参照して下さい。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD（`opc1 == '0x' && opc2 == '00'` の場合）
アドバンスト SIMD（その他の場合）

```
VMOV<c>.<dt> <Rt>, <Dn[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	U	opc1		1	Vn				Rt				1	0	1	1	N	opc2		1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	U	opc1		1	Vn				Rt				1	0	1	1	N	opc2		1	(0)	(0)	(0)	(0)

```
case U:opc1:opc2 of
    when ‘x1xxx’  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when ‘x0xx1’  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when ‘00x00’  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when ‘10x00’  UNDEFINED;
    when ‘x0x10’  UNDEFINED;
t = UInt(Rt);  n = UInt(N:Vn);  unsigned = (U == ‘1’);
if t == 15 || (CurrentInstrSet() != InstrSet_ARM && t == 13) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VMOV<c>{.<dt>}  <Rt>, <Dn[x]>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<dt> The data type. It must be one of:

- S8 Encoded as opc1<2:1> = '01'. [x] is encoded in opc1<0>, opc2.
- S16 Encoded as opc1<2:1>, opc2<0> = '001'. [x] is encoded in opc1<0>, opc2<1>.
- U8 Encoded as opc1<2:1> = '11'. [x] is encoded in opc1<0>, opc2.
- U16 Encoded as opc1<2:1>, opc2<0> = '101.' [x] is encoded in opc1<0>, opc2<1>.
- 32 Encoded as opc1<2:1>, opc2<1:0> = '0000'. [x] is encoded in opc1<0>.
- **omitted** equivalent to 32.

<Dm[x]> The scalar. For details of how [x] is encoded see the description of <dt>.

<Rt> The destination ARM core register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if unsigned then
        R[t] = ZeroExtend(Elem[D[n+r],index,esize]);
    else
        R[t] = SignExtend(Elem[D[n+r],index,esize]);
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VMOV<c>{.<dt>}  <Rt>, <Dn[x]>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<dt> データ型。次のいずれかを使用できます。

- S8 opc1〈2:1〉= '01' としてエンコードされます。[x] は、opc1〈0〉、opc2 にエンコードされます。
- S16 opc1〈2:1〉、opc2〈0〉= '001' としてエンコードされます。[x] は、opc1〈0〉、opc2〈1〉にエンコードされます。
- U8 opc1〈2:1〉= '11' としてエンコードされます。[x] は、opc1〈0〉、opc2 にエンコードされます。
- U16 opc1〈2:1〉、opc2〈0〉= '101' としてエンコードされます。[x] は、opc1〈0〉、opc2〈1〉にエンコードされます。
- 32 opc1〈2:1〉、opc2〈1:0〉= '0000' としてエンコードされます。[x] は、opc1〈0〉にエンコードされます。
- **省略** 32 と等価です。

<Dm[x]> スカラ。[x] のエンコード方法の詳細については、<dt> の説明を参照して下さい。

<Rt> デスティネーション ARM コアレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if unsigned then
        R[t] = ZeroExtend(Elem[D[n+r],index,esize]);
    else
        R[t] = SignExtend(Elem[D[n+r],index,esize]);
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.330 VMOV (between ARM core register and single-precision register)

This instruction transfers the contents of a single-precision VFP register to an ARM core register, or the contents of an ARM core register to a single-precision VFP register.

Encoding T1 / A1 VFPv2, VFPv3

```
VMOV<c> <Sn>, <Rt>
VMOV<c> <Rt>, <Sn>
```

15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
1 1 1 0	1 1 1 0	0 0 0	op	Vn	Rt	1 0 1 0	N	(0) (0)	1	(0) (0) (0) (0)

31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	1 1 1 0	0 0 0	op	Vn	Rt	1 0 1 0	N	(0) (0)	1	(0) (0) (0) (0)

```
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 || (CurrentInstrSet() != InstrSet_ARM && t == 13) then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.330 VMOV（ARM コアレジスタと単精度レジスタとの間の転送）

VMOV（ARM コアレジスタと単精度レジスタとの間の転送）は、単精度 VFP レジスタの内容を ARM コアレジスタに転送するか、または ARM コアレジスタの内容を単精度 VFP レジスタに転送します。

エンコード T1/A1 VFPv2、VFPv3

```
VMOV<c> <Sn>, <Rt>
VMOV<c> <Rt>, <Sn>
```

15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
1 1 1 0	1 1 1 0	0 0 0	op	Vn	Rt	1 0 1 0	N	(0) (0)	1	(0) (0) (0) (0)

31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
cond	1 1 1 0	0 0 0	op	Vn	Rt	1 0 1 0	N	(0) (0)	1	(0) (0) (0) (0)

```
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 || (CurrentInstrSet() != InstrSet_ARM && t == 13) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VMOV<c><q>  <Sn>, <Rt>`	Encoded as op = 0
`VMOV<c><q>  <Rt>, <Sn>`	Encoded as op = 1

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Sn>` The single-precision VFP register.

`<Rt>` The ARM core register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_register then
        R[t] = S[n];
    else
        S[n] = R[t];
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VMOV<c><q>  <Sn>, <Rt>`	op = 0 としてエンコードされる
`VMOV<c><q>  <Rt>, <Sn>`	op = 1 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Sn>` 単精度 VFP レジスタ

`<Rt>` ARM コアレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_register then
        R[t] = S[n];
    else
        S[n] = R[t];
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.331 VMOV (between two ARM core registers and two single-precision registers)

This instruction transfers the contents of two consecutively numbered single-precision VFP registers to two ARM core registers, or the contents of two ARM core registers to a pair of single-precision VFP registers. The ARM core registers do not have to be contiguous.

Encoding T1 / A1 VFPv2, VFPv3

```
VMOV<c> <Sm>, <Sm1>, <Rt>, <Rt2>
VMOV<c> <Rt>, <Rt2>, <Sm>, <Sm1>
```

15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
1 1 1 0	1 1 0 0	0 1 0	op	Rt2	Rt	1 0 1 0	0 0	M	1	Vm

31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	1 1 0 0	0 1 0	op	Rt2	Rt	1 0 1 0	0 0	M	1	Vm

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(Vm:M);
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if CurrentInstrSet() != InstrSet_ARM && (t == 13 || t2 == 13) then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.331 VMOV（2 つの ARM コアレジスタと 2 つの単精度レジスタとの間の転送）

VMOV（2 つの ARM コアレジスタと 2 つの単精度レジスタとの間の転送）は、番号が連続している 2 つの単精度 VFP レジスタの内容を 2 つの ARM コアレジスタに転送するか、または 2 つの ARM コアレジスタの内容を単精度 VFP レジスタのペアに転送します。ARM コアレジスタは連続している必要はありません。

エンコード T1/A1 VFPv2、VFPv3

```
VMOV<c> <Sm>, <Sm1>, <Rt>, <Rt2>
VMOV<c> <Rt>, <Rt2>, <Sm>, <Sm1>
```

15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
1 1 1 0	1 1 0 0	0 1 0	op	Rt2	Rt	1 0 1 0	0 0	M	1	Vm

31 30 29 28	27 26 25 24	23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6	5	4	3 2 1 0
cond	1 1 0 0	0 1 0	op	Rt2	Rt	1 0 1 0	0 0	M	1	Vm

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(Vm:M);
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if CurrentInstrSet() != InstrSet_ARM && (t == 13 || t2 == 13) then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VMOV<c><q>  <Sm>, <Sm1>, <Rt>, <Rt2>`	Encoded as op = 0
`VMOV<c><q>  <Rt>, <Rt2>, <Sm>, <Sm1>`	Encoded as op = 1

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Sm>` The first single-precision VFP register.

`<Sm1>` The second single-precision VFP register. This is the next single-precision VFP register after `<Sm>`.

`<Rt>` The ARM core register that `<Sm>` is transferred to or from.

`<Rt2>` The ARM core register that `<Sm1>` is transferred to or from.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R[t] = S[m];
        R[t2] = S[m+1];
    else
        S[m] = R[t];
        S[m+1] = R[t2];
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`VMOV<c><q>  <Sm>, <Sm1>, <Rt>, <Rt2>`	op = 0 としてエンコードされる
`VMOV<c><q>  <Rt>, <Rt2>, <Sm>, <Sm1>`	op = 1 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Sm>` 最初の単精度 VFP レジスタ

`<Sm1>` 2 番目の単精度 VFP レジスタ。これは、`<Sm>` の次の単精度 VFP レジスタです。

`<Rt>` `<Sm>` の転送先または転送元となる ARM コアレジスタ

`<Rt2>` `<Sm1>` の転送先または転送元となる ARM コアレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R[t] = S[m];
        R[t2] = S[m+1];
    else
        S[m] = R[t];
        S[m+1] = R[t2];
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.332 VMOV (between two ARM core registers and a doubleword extension register)

This instruction copies two words from two ARM core registers into a doubleword extension register, or from a doubleword extension register to two ARM core registers.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD

```
VMOV<c> <Dm>, <Rt>, <Rt2>
VMOV<c> <Rt>, <Rt2>, <Dm>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0	1 1 0 0	0 1 0 op	Rt2	Rt	1 0 1 1	0 0 M 1	Vm

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	1 1 0 0	0 1 0 op	Rt2	Rt	1 0 1 1	0 0 M 1	Vm

```
to_arm_registers = (op == '1');  t = UInt(Rd);  t2 = UInt(Rt2);  m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if CurrentInstrSet() != InstrSet_ARM && (t == 13 || t2 == 13) then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.332 VMOV（2 つの ARM コアレジスタとダブルワード拡張機能レジスタとの間のコピー）

VMOV（2 つの ARM コアレジスタとダブルワード拡張機能レジスタとの間のコピー）は、2 つのワードを 2 つの ARM コアレジスタからダブルワード拡張機能レジスタにコピーするか、またはダブルワード拡張機能レジスタから 2 つの ARM コアレジスタにコピーします。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD

```
VMOV<c> <Dm>, <Rt>, <Rt2>
VMOV<c> <Rt>, <Rt2>, <Dm>
```

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0	1 1 0 0	0 1 0 op	Rt2	Rt	1 0 1 1	0 0 M 1	Vm

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	1 1 0 0	0 1 0 op	Rt2	Rt	1 0 1 1	0 0 M 1	Vm

```
to_arm_registers = (op == '1');  t = UInt(Rd);  t2 = UInt(Rt2);  m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if CurrentInstrSet() != InstrSet_ARM && (t == 13 || t2 == 13) then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VMOV<c><q>  <Dm>, <Rt>, <Rt2>`	Encoded as op = 0
`VMOV<c><q>  <Rt>, <Rt2>, <Dm>`	Encoded as op = 1

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Dm>` The doubleword extension register.

`<Rt>, <Rt2>` The two ARM core registers.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R[t] = D[m]<31:0>;
        R[t2] = D[m]<63:32>;
    else
        D[m]<31:0> = R[t];
        D[m]<63:32> = R[t2];
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VMOV<c><q>  <Dm>, <Rt>, <Rt2>`	op = 0 としてエンコードされる
`VMOV<c><q>  <Rt>, <Rt2>, <Dm>`	op = 1 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Dm>` ダブルワード拡張機能レジスタ

`<Rt>, <Rt2>` 2 つの ARM コアレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R[t] = D[m]<31:0>;
        R[t2] = D[m]<63:32>;
    else
        D[m]<31:0> = R[t];
        D[m]<63:32> = R[t2];
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.333 VMOVL

Vector Move Long takes each element in a doubleword vector, sign or zero-extends them to twice their original length, and places the results in a quadword vector.

Encoding T1 / A1 Advanced SIMD

```
VMOVL<c>.<dt> <Qd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm3			0	0	0	Vd				1	0	1	0	0	0	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm3			0	0	0	Vd				1	0	1	0	0	0	M	1	Vm			

```
if imm3 == '000' then SEE "Related encodings";
if imm3 != '001' && imm3 != '010' && imm3 != '100' then SEE VSHLL;
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3);
unsigned = (U == '1');  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.333 VMOVL

VMOVL（ベクタロング移動）は、ダブルワードベクタの各エレメントを取得して符号拡張またはゼロ拡張し、元の長さを 2 倍にし、結果をクワッドワードベクタに配置します。

エンコード T1/A1 アドバンスト SIMD

```
VMOVL<c>.<dt> <Qd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm3			0	0	0	Vd				1	0	1	0	0	0	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm3			0	0	0	Vd				1	0	1	0	0	0	M	1	Vm			

```
if imm3 == '000' then SEE "Related encodings";
if imm3 != '001' && imm3 != '010' && imm3 != '100' then SEE VSHLL;
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3);
unsigned = (U == '1');  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VMOVL<c><q>.dt  <Qd>, <Dm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VMOVL` instruction must be unconditional.

`<dt>` The data type for the elements of the operand. It must be one of:

- `S8` encoded as U = 0, imm3 = '001'
- `S16` encoded as U = 0, imm3 = '010'
- `S32` encoded as U = 0, imm3 = '100'
- `U8` encoded as U = 1, imm3 = '001'
- `U16` encoded as U = 1, imm3 = '010'
- `U32` encoded as U = 1, imm3 = '100'.

`<Qd>, <Dm>` The destination vector and the operand vector.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Int(Elem[D[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
VMOVL<c><q>.dt  <Qd>, <Dm>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VMOVL` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S8` U = 0、imm3 = '001' としてエンコードされます。
- `S16` U = 0、imm3 = '010' としてエンコードされます。
- `S32` U = 0、imm3 = '100' としてエンコードされます。
- `U8` U = 1、imm3 = '001' としてエンコードされます。
- `U16` U = 1、imm3 = '010' としてエンコードされます。
- `U32` U = 1、imm3 = '100' としてエンコードされます。

`<Qd>, <Dm>` デスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Int(Elem[D[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.334 VMOVN

Vector Move and Narrow copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

The operand vector elements can be any one of 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Encoding T1 / A1 Advanced SIMD

```
VMOVN<c>.<dt> <Dd>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	1	0	0	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	1	0	0	0	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.334 VMOVN

VMOVN（ベクタ移動および縮小）は、クワッドワードベクタの各エレメントの下位半分を、ダブルワードベクタの対応するエレメントにコピーします。

オペランドベクタのエレメントには、16 ビット、32 ビット、または 64 ビットの整数を使用できます。整数には符号付きと符号なしの区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VMOVN<c>.<dt> <Dd>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	1	0	0	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	1	0	0	0	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VMOVN<c><q>.<dt>  <Dd>, <Qm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7. An ARM VMOVN instruction must be unconditional.

<dt> The data type for the elements of the operand. It must be one of:

- I16 encoded as size = 0b00
- I32 encoded as size = 0b01
- I64 encoded as size = 0b10.

<Dd>, <Qm> The destination vector and the operand vector.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        Elem[D[d],e,esize] = Elem[Q[m>>1],e,2*esize]<esize-1:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VMOVN<c><q>.<dt>  <Dd>, <Qm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM VMOVN は常に無条件命令です。

<dt> オペランドのエレメントのデータ型。次のいずれかを使用できます。

- I16 size = 0b00 としてエンコードされます。
- I32 size = 0b01 としてエンコードされます。
- I64 size = 0b10 としてエンコードされます。

<Dd>, <Qm> デスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        Elem[D[d],e,esize] = Elem[Q[m>>1],e,2*esize]<esize-1:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.335 VMRS

Move to ARM core register from Advanced SIMD and VFP extension System Register moves the value of the FPSCR to a general-purpose register.

For details of system level use of this instruction, see *VMRS* on page B6-27.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD

```
VMRS<c> <Rt>, FPSCR
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	1	0	0	0	1	Rt				1	0	1	0	0	(0)	(0)	1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	1	1	1	0	0	0	1	Rt				1	0	1	0	0	(0)	(0)	1	(0)	(0)	(0)	(0)

```
t = UInt(Rt);
if t == 13 && CurrentInstrSet() != InstrSet_ARM then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.335 VMRS

VMRS（アドバンスト SIMD および VFP 拡張機能システムレジスタから ARM コアレジスタへの移動）は、FPSCR の値を汎用レジスタに移動します。

この命令をシステムレベルで使用する方法の詳細については、*VMRS* P. B6-27 を参照して下さい。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD

```
VMRS<c> <Rt>, FPSCR
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	1	0	0	0	1	Rt				1	0	1	0	0	(0)	(0)	1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	1	1	1	0	0	0	1	Rt				1	0	1	0	0	(0)	(0)	1	(0)	(0)	(0)	(0)

```
t = UInt(Rt);
if t == 13 && CurrentInstrSet() != InstrSet_ARM then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VMRS<c><q>  <Rt>, FPSCR
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rt> The destination ARM core register. This register can be R0-R14 or APSR_nzcv. APSR_nzcv is encoded as Rt = '1111', and the instruction transfers the FPSCR N, Z, C, and V flags to the APSR N, Z, C, and V flags.

The pre-UAL instruction `FMSTAT` is equivalent to `VMRS APSR_nzcv, FPSCR`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    SerializeVFP();  VFPExcBarrier();
    if t != 15 then
        R[t] = FPSCR;
    else
        APSR.N = FPSCR.N;
        APSR.Z = FPSCR.Z;
        APSR.C = FPSCR.C;
        APSR.V = FPSCR.V;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VMRS<c><q>  <Rt>, FPSCR
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rt> デスティネーション ARM コアレジスタ。R0 ～ R14 または APSR_nzcv を指定できます。APSR_nzcv は Rt = '1111' としてエンコードされます。この命令は、FPSCR の N、Z、C、V フラグを APSR の N、Z、C、V フラグに転送します。

UAL 以前の命令の `FMSTAT` は、`VMRS APSR_nzcv, FPSCR` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    SerializeVFP();  VFPExcBarrier();
    if t != 15 then
        R[t] = FPSCR;
    else
        APSR.N = FPSCR.N;
        APSR.Z = FPSCR.Z;
        APSR.C = FPSCR.C;
        APSR.V = FPSCR.V;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.336 VMSR

Move to Advanced SIMD and VFP extension System Register from ARM core register moves the value of a general-purpose register to the FPSCR.

For details of system level use of this instruction, see *VMSR* on page B6-29.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD

```
VMSR<c> FPSCR, <Rt>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	0	0	0	0	1	Rt				1	0	1	0	0	(0)	(0)	1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	1	1	0	0	0	0	1	Rt				1	0	1	0	0	(0)	(0)	1	(0)	(0)	(0)	(0)

```
t = UInt(Rt);
if t == 15 || (t == 13 && CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.336 VMSR

VMRS（ARM コアレジスタからアドバンスト SIMD および VFP 拡張機能システムレジスタへの移動）は、汎用レジスタの値を FPSCR に移動します。

この命令をシステムレベルで使用する方法の詳細については、*VMSR* P. B6-29 を参照して下さい。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD

```
VMSR<c> FPSCR, <Rt>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	0	0	0	0	1	Rt				1	0	1	0	0	(0)	(0)	1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	1	1	0	0	0	0	1	Rt				1	0	1	0	0	(0)	(0)	1	(0)	(0)	(0)	(0)

```
t = UInt(Rt);
if t == 15 || (t == 13 && CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VMSR<c><q>  FPSCR, <Rt>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rt> The general-purpose register to be transferred to the FPSCR.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    SerializeVFP();  VFPExcBarrier();
    FPSCR = R[t];
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VMSR<c><q>  FPSCR, <Rt>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rt> FPSCR に転送される汎用レジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    SerializeVFP();  VFPExcBarrier();
    FPSCR = R[t];
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.337 VMUL, VMULL (integer and polynomial)

Vector Multiply multiplies corresponding elements in two vectors. Vector Multiply Long does the same thing, but with destination vector elements that are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over {0,1}* on page A2-67.

Encoding T1 / A1 Advanced SIMD

```
VMUL<c>.<dt> <Qd>, <Qn>, <Qm>
VMUL<c>.<dt> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	op	1	1	1	1	0	D	size		Vn				Vd				1	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	op	0	D	size		Vn				Vd				1	0	0	1	N	Q	M	1	Vm			

```
if size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
polynomial = (op == '1');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 Advanced SIMD

```
VMULL<c>.<dt> <Qd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				1	1	op	0	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				1	1	op	0	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if op == '1' && (U != '0' || size != '00') then UNDEFINED;
if Vd<0> == '1' then UNDEFINED;
polynomial = (op == '1');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.337 VMUL、VMULL（整数および多項式）

VMUL（ベクタ乗算（整数および多項式））は、2 つのベクタの対応するエレメントを乗算します。VMULL（ベクタロング乗算（整数および多項式））は、これと同じ処理を実行しますが、デスティネーションベクタのエレメントの長さが、乗算されるエレメントの 2 倍の長さです。

多項式の乗算の詳細については、*{0,1} における多項式算術演算* P. A2-67 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VMUL<c>.<dt> <Qd>, <Qn>, <Qm>
VMUL<c>.<dt> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	op	1	1	1	1	0	D	size		Vn				Vd				1	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	op	0	D	size		Vn				Vd				1	0	0	1	N	Q	M	1	Vm			

```
if size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
polynomial = (op == '1');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 アドバンスト SIMD

```
VMULL<c>.<dt> <Qd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				1	1	op	0	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				1	1	op	0	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if op == '1' && (U != '0' || size != '00') then UNDEFINED;
if Vd<0> == '1' then UNDEFINED;
polynomial = (op == '1');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VMUL<c><q>.<type><size>  {<Qd>,} <Qn>, <Qm>`	Encoding T1 / A1. Q = 1
`VMUL<c><q>.<type><size>  {<Dd>,} <Dn>, <Dm>`	Encoding T1 / A1. Q = 0
`VMULL<c><q>.<type><size>  <Qd>, <Dn>, <Dm>`	Encoding T2 / A2

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VMUL` or `VMULL` instruction must be unconditional.

`<type>` The data type for the elements of the operands. It must be one of:

- `S` op = 0 in both encodings. U = 0 in encoding T2 / A2
- `U` op = 0 in both encodings. U = 1 in encoding T2 / A2
- `I` op = 0 in encoding T1 / A1, not available in encoding T2 / A2
- `P` op = 1 in both encodings. U= 0 in encoding T2 / A2.

When `<type>` is `P`, `<size>` must be `8`.

`<size>` The data size for the elements of the operands. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

`<Qd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a long operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op1val = Int(op1, unsigned);
            op2 = Elem[D[m+r],e,esize];  op2val = Int(op2, unsigned);
            if polynomial then
                product = PolynomialMult(op1,op2);
            else
                product = (op1val*op2val)<2*esize-1:0>;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = product;
            else
                Elem[D[d+r],e,esize] = product<esize-1:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VMUL<c><q>.<type><size>  {<Qd>,} <Qn>, <Qm>`	エンコード T1/A1。Q = 1
`VMUL<c><q>.<type><size>  {<Dd>,} <Dn>, <Dm>`	エンコード T1/A1。Q = 0
`VMULL<c><q>.<type><size>  <Qd>, <Dn>, <Dm>`	エンコード T2/A2

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VMUL` または `VMULL` は常に無条件命令です。

`<type>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S` 両方のエンコードで op = 0。エンコード T2/A2 では U = 0。
- `U` 両方のエンコードで op = 0。エンコード T2/A2 では U = 1。
- `I` エンコード T1/A1 では op = 0。エンコード T2/A2 では使用できません。
- `P` 両方のエンコードで op = 1。エンコード T2/A2 では U = 0。

`<type>` が `P` の場合、`<size>` を `8` にする必要があります。

`<size>` オペランドのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Qd>, <Dn>, <Dm>` ロング演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op1val = Int(op1, unsigned);
            op2 = Elem[D[m+r],e,esize];  op2val = Int(op2, unsigned);
            if polynomial then
                product = PolynomialMult(op1,op2);
            else
                product = (op1val*op2val)<2*esize-1:0>;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = product;
            else
                Elem[D[d+r],e,esize] = product<esize-1:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.338 VMUL (floating-point)

Vector Multiply multiplies corresponding elements in two vectors, and places the results in the destination vector. Vector Multiply Long does the same thing, but with destination vector elements that are twice as long as the elements that are multiplied.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
VMUL<c>.F32 <Qd>, <Qn>, <Qm>
VMUL<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VMUL<c>.F64 <Dd>, <Dn>, <Dm>
VMUL<c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	1	0	Vn				Vd				1	0	1	sz	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	D	1	0	Vn				Vd				1	0	1	sz	N	0	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP vectors Encoding T2 / A2 can operate on VFP vectors under control of the FPSCR.LEN and FPSCR.STRIDE bits. For details see Appendix F *VFP Vector Operation Support*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.338 VMUL、VMULL（浮動小数点数）

VMUL（ベクタ乗算（浮動小数点数））は、2 つのベクタの対応するエレメントを乗算し、結果をデスティネーションベクタに配置します。VMULL（ベクタロング乗算（浮動小数点数））は、これと同じ処理を実行しますが、デスティネーションベクタのエレメントの長さが、乗算されるエレメントの 2 倍の長さです。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VMUL<c>.F32 <Qd>, <Qn>, <Qm>
VMUL<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VMUL<c>.F64 <Dd>, <Dn>, <Dm>
VMUL<c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	1	0	Vn				Vd				1	0	1	sz	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	D	1	0	Vn				Vd				1	0	1	sz	N	0	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP ベクタ エンコード T2/A2 は、FPSCR.LEN ビットと FPSCR.STRIDE ビットの制御の下で VFP ベクタに作用することができます。詳細については、付録 F *「VFP ベクタ演算のサポート」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VMUL<c><q>.F32 {<Qd>,} <Qn>, <Qm>`	Encoding T1 / A1, Q = 1, sz = 0
`VMUL<c><q>.F32 {<Dd>,} <Dn>, <Dm>`	Encoding T1 / A1, Q = 0, sz = 0
`VMUL<c><q>.F64 {<Dd>,} <Dn>, <Dm>`	Encoding T2 / A2, sz = 1
`VMUL<c><q>.F32 {<Sd>,} <Sn>, <Sm>`	Encoding T2 / A2, sz = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VMUL` instruction must be unconditional.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

`<Sd>, <Sn>, <Sm>` The destination vector and the operand vectors, for a singleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], FALSE);
    else            // VFP instruction
        if dp_operation then
            D[d] = FPMul(D[n], D[m], TRUE);
        else
            S[d] = FPMul(S[n], S[m], TRUE);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, Overflow, Underflow, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VMUL<c><q>.F32 {<Qd>,} <Qn>, <Qm>`	エンコード T1/A1、Q = 1、sz = 0
`VMUL<c><q>.F32 {<Dd>,} <Dn>, <Dm>`	エンコード T1/A1、Q = 0、sz = 0
`VMUL<c><q>.F64 {<Dd>,} <Dn>, <Dm>`	エンコード T2/A2、sz = 1
`VMUL<c><q>.F32 {<Sd>,} <Sn>, <Sm>`	エンコード T2/A2、sz = 0

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VMUL` は常に無条件命令です。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Sd>, <Sn>, <Sm>` シングルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], FALSE);
    else            // VFP instruction
        if dp_operation then
            D[d] = FPMul(D[n], D[m], TRUE);
        else
            S[d] = FPMul(S[n], S[m], TRUE);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、オーバフロー、アンダーフロー、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.339 VMUL, VMULL (by scalar)

Vector Multiply multiplies each element in a vector by a scalar, and places the results in a second vector. Vector Multiply Long does the same thing, but with destination vector elements that are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars* on page A7-9.

Encoding T1 / A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

```
VMUL<c>.<dt> <Qd>, <Qn>, <Dm[x]>
VMUL<c>.<dt> <Dd>, <Dn>, <Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	size		Vn				Vd				1	0	0	F	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	size		Vn				Vd				1	0	0	F	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Encoding T2 / A2 Advanced SIMD

```
VMULL<c>.<dt> <Qd>, <Dn>, <Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				1	0	1	0	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				1	0	1	0	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;  floating_point = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.339 VMUL、VMULL（スカラ）

VMUL（ベクタ乗算（スカラ））は、ベクタの各エレメントとスカラを乗算し、結果を 2 番目のベクタに配置します。VMULL（ベクタロング乗算（スカラ））は、これと同じ処理を実行しますが、デスティネーションベクタのエレメントの長さが、乗算されるエレメントの 2 倍の長さです。

スカラの詳細については、*アドバンスト SIMD スカラ* P. A7-9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは F = 1 は未定義）

```
VMUL<c>.<dt> <Qd>, <Qn>, <Dm[x]>
VMUL<c>.<dt> <Dd>, <Dn>, <Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	size		Vn				Vd				1	0	0	F	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	size		Vn				Vd				1	0	0	F	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

エンコード T2/A2 アドバンスト SIMD

```
VMULL<c>.<dt> <Qd>, <Dn>, <Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				1	0	1	0	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				1	0	1	0	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;  floating_point = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VMUL<c><q>.<dt> {<Qd>,} <Qn>, <Dm[x]>`	Encoding T1 / A1, Q = 1
`VMUL<c><q>.<dt> {<Dd>,} <Dn>, <Dm[x]>`	Encoding T1 / A1, Q = 0
`VMULL<c><q>.<dt> <Qd>, <Dn>, <Dm[x]>`	Encoding T2 / A2

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VMUL` or `VMULL` instruction must be unconditional.

`<dt>` The data type for the scalar, and the elements of the operand vector. It must be one of:

- `I16` encoding T1 / A1, size = 0b01, F = 0
- `I32` encoding T1 / A1, size = 0b10, F = 0
- `F32` encoding T1 / A1, size = 0b10, F = 1
- `S16` encoding T2 / A2, size = 0b01, U = 0
- `S32` encoding T2 / A2, size = 0b10, U = 0
- `U16` encoding T2 / A2, size = 0b01, U = 1
- `U32` encoding T2 / A2, size = 0b10, U = 1.

`<Qd>, <Qn>` The destination vector, and the operand vector, for a quadword operation.

`<Dd>, <Dn>` The destination vector, and the operand vector, for a doubleword operation.

`<Qd>, <Dn>` The destination vector, and the operand vector, for a long operation.

`<Dm[x]>` The scalar. `Dm` is restricted to D0-D7 if `<dt>` is `I16`, `S16`, or `U16`, or D0-D15 otherwise.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[D[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, FALSE);
            else
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
                else
                    Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, Overflow, Underflow, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`VMUL<c><q>.<dt> {<Qd>,} <Qn>, <Dm[x]>`	エンコード T1/A1、Q = 1
`VMUL<c><q>.<dt> {<Dd>,} <Dn>, <Dm[x]>`	エンコード T1/A1、Q = 0
`VMULL<c><q>.<dt> <Qd>, <Dn>, <Dm[x]>`	エンコード T2/A2

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VMUL` または `VMULL` は常に無条件命令です。

`<dt>` スカラと、オペランドベクタのエレメントのデータ型。次のいずれかを使用できます。

- `I16` エンコード T1/A1、size = 0b01、F = 0
- `I32` エンコード T1/A1、size = 0b10、F = 0
- `F32` エンコード T1/A1、size = 0b10、F = 1
- `S16` エンコード T2/A2、size = 0b01、U = 0
- `S32` エンコード T2/A2、size = 0b10、U = 0
- `U16` エンコード T2/A2、size = 0b01、U = 1
- `U32` エンコード T2/A2、size = 0b10、U = 1

`<Qd>, <Qn>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Qd>, <Dn>` ロング演算の場合のデスティネーションベクタとオペランドベクタ

`<Dm[x]>` スカラ。`<dt>` が `I16`、`S16`、または `U16` の場合は `Dm` は D0 ～ D7 に限定され、それ以外の場合は D0 ～ D15 に限定されます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[D[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, FALSE);
            else
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
                else
                    Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、オーバフロー、アンダーフロー、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.340 VMVN (immediate)

Vector Bitwise NOT (immediate) places the bitwise inverse of an immediate integer constant into every element of the destination register. For the range of constants available, see *One register and a modified immediate value* on page A7-21.

Encoding T1 / A1 Advanced SIMD

```
VMVN<c>.<dt> <Qd>, #<imm>
VMVN<c>.<dt> <Dd>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd				cmode				0	Q	1	1	imm4			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd				cmode				0	Q	1	1	imm4			

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.340 VMVN（イミディエート）

VMVN（ベクタビット単位否定（イミディエート））は、イミディエート整数定数のビット単位の否定を、デスティネーションレジスタのすべてのエレメントに配置します。使用可能な定数の範囲については、*1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VMVN<c>.<dt> <Qd>, #<imm>
VMVN<c>.<dt> <Dd>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd				cmode				0	Q	1	1	imm4			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd				cmode				0	Q	1	1	imm4			

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VMVN<c><q>.dt  <Qd>, #<imm>`	Encoding T1 / A1, Q = 1
`VMVN<c><q>.dt  <Dd>, #<imm>`	Encoding T1 / A1, Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VMVN` instruction must be unconditional.

`<dt>` The data type. It must be either `I16` or `I32`.

`<Qd>` The destination register for a quadword operation.

`<Dd>` The destination register for a doubleword operation.

`<imm>` A constant of the specified type.

See *One register and a modified immediate value* on page A7-21 for the range of constants available, and the encoding of `<dt>` and `<imm>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(imm64);
```

Exceptions

Undefined Instruction.

Pseudo-instructions

One register and a modified immediate value on page A7-21 describes pseudo-instructions with a combination of `<dt>` and `<imm>` that is not supported by hardware, but that generates the same destination register value as a different combination that is supported by hardware.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VMVN<c><q>.dt  <Qd>, #<imm>`	エンコード T1/A1、Q = 1
`VMVN<c><q>.dt  <Dd>, #<imm>`	エンコード T1/A1、Q = 0

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VMVN` は常に無条件命令です。

`<dt>` データ型。`I16` または `I32` の必要があります。

`<Qd>` クワッドワード演算の場合のデスティネーションレジスタ

`<Dd>` ダブルワード演算の場合のデスティネーションレジスタ

`<imm>` 指定された型の定数。

使用可能な定数の範囲と、`<dt>` および `<imm>` のエンコードの詳細については、*1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(imm64);
```

例外

未定義命令

擬似命令

ハードウェアでサポートされていないにもかかわらず、ハードウェアでサポートされている別の組み合わせと同じデスティネーションレジスタの値を生成する、`<dt>` と `<imm>` の組み合わせを持つ擬似命令については、*1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.341 VMVN (register)

Vector Bitwise NOT (register) takes a value from a register, inverts the value of each bit, and places the result in the destination register. The registers can be either doubleword or quadword.

Encoding T1 / A1 Advanced SIMD

```
VMVN<c> <Qd>, <Qm>
VMVN<c> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	0	1	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	0	1	1	Q	M	0	Vm			

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.341 VMVN（レジスタ）

VMVN（ベクタビット単位否定（レジスタ））は、レジスタから値を取得し、各ビットの値を反転し、結果をデスティネーションレジスタに配置します。レジスタには、ダブルワードまたはクワッドワードを使用できます。

エンコード T1/A1 アドバンスト SIMD

```
VMVN<c> <Qd>, <Qm>
VMVN<c> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	0	1	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	0	1	1	Q	M	0	Vm			

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VMVN<c><q>{.<dt>}  <Qd>, <Qm>
VMVN<c><q>{.<dt>}  <Dd>, <Dm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM `VMVN` instruction must be unconditional.
`<dt>`	An optional data type. It is ignored by assemblers, and does not affect the encoding.
`<Qd>, <Qm>`	The destination vector and the operand vector, for a quadword operation.
`<Dd>, <Dm>`	The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(D[m+r]);
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VMVN<c><q>{.<dt>}  <Qd>, <Qm>
VMVN<c><q>{.<dt>}  <Dd>, <Dm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VMVN` は常に無条件命令です。
`<dt>`	オプションのデータ型。アセンブラによって無視され、エンコードに影響を与えません。
`<Qd>, <Qm>`	クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Dd>, <Dm>`	ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(D[m+r]);
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.342 VNEG

Vector Negate negates each element in a vector, and places the results in a second vector. The floating-point version only inverts the sign bit.

Encoding T1 / A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

```
VNEG<c>.<dt> <Qd>, <Qm>
VNEG<c>.<dt> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	1	1	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	1	1	1	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VNEG<c>.F64 <Dd>, <Dm>
VNEG<c>.F32 <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	1	Vd				1	0	1	sz	0	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	0	0	1	Vd				1	0	1	sz	0	1	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP vectors Encoding T2 / A2 can operate on VFP vectors under control of the FPSCR.LEN and FPSCR.STRIDE bits. For details see Appendix F *VFP Vector Operation Support*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.342 VNEG

VNEG（ベクタ否定）は、ベクタの各エレメントを反転し、結果を 2 番目のベクタに配置します。浮動小数点バージョンは、符号ビットの反転のみを行います。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは F = 1 は未定義）

```
VNEG<c>.<dt> <Qd>, <Qm>
VNEG<c>.<dt> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	F	1	1	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	F	1	1	1	Q	M	0	Vm			

```
if size == '11' || (F == '1' && size != '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VNEG<c>.F64 <Dd>, <Dm>
VNEG<c>.F32 <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	1	Vd				1	0	1	sz	0	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	0	0	1	Vd				1	0	1	sz	0	1	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP ベクタ エンコード T2/A2 は、FPSCR.LEN ビットと FPSCR.STRIDE ビットの制御の下で VFP ベクタに作用することができます。詳細については、付録 F *「VFP ベクタ演算のサポート」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VNEG<c><q>.<dt> <Qd>, <Qm>`	`<dt> != F64`
`VNEG<c><q>.<dt> <Dd>, <Dm>`	
`VNEG<c><q>.F32 <Sd>, <Sm>`	VFP only, encoding T2/A2, sz = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VNEG` instruction must be unconditional.

`<dt>` The data type for the elements of the vectors. It must be one of:

- `S8` encoding T1 / A1, size = 0b00, F = 0
- `S16` encoding T1 / A1, size = 0b01, F = 0
- `S32` encoding T1 / A1, size = 0b10, F = 0
- `F32` encoding T1 / A1, size = 0b10, F = 1
- `F64` encoding T2 / A2, sz = 1.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

`<Sd>, <Sm>` The destination vector and the operand vector, for a singleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPNeg(Elem[D[m+r],e,esize]);
                else
                    result = -SInt(Elem[D[m+r],e,esize]);
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
    else             // VFP instruction
        if dp_operation then
            D[d] = FPNeg(D[m]);
        else
            S[d] = FPNeg(S[m]);
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`VNEG<c><q>.<dt> <Qd>, <Qm>`	`<dt> != F64`
`VNEG<c><q>.<dt> <Dd>, <Dm>`	
`VNEG<c><q>.F32 <Sd>, <Sm>`	VFP のみ、エンコード T2/A2、sz = 0

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VNEG` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S8` エンコード T1/A1、size = 0b00、F = 0
- `S16` エンコード T1/A1、size = 0b01、F = 0
- `S32` エンコード T1/A1、size = 0b10、F = 0
- `F32` エンコード T1/A1、size = 0b10、F = 1
- `F64` エンコード T2/A2、sz = 1

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Sd>, <Sm>` シングルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPNeg(Elem[D[m+r],e,esize]);
                else
                    result = -SInt(Elem[D[m+r],e,esize]);
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
    else             // VFP instruction
        if dp_operation then
            D[d] = FPNeg(D[m]);
        else
            S[d] = FPNeg(S[m]);
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.343 VNMLA, VNMLS, VNMUL

VNMLA multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the negation of the product, and writes the result back to the destination register.

VNMLS multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register.

VNMUL multiplies together two floating-point register values, and writes the negation of the result to the destination register.

Encoding T1 / A1 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VNMLA<c>.F64 <Dd>, <Dn>, <Dm>
VNMLA<c>.F32 <Sd>, <Sn>, <Sm>
VNMLS<c>.F64 <Dd>, <Dn>, <Dm>
VNMLS<c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	0	1	Vn				Vd				1	0	1	sz	N	op	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	D	0	1	Vn				Vd				1	0	1	sz	N	op	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
type = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

Encoding T2 / A2 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VNMUL<c>.F64 <Dd>, <Dn>, <Dm>
VNMUL<c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	1	0	Vn				Vd				1	0	1	sz	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	D	1	0	Vn				Vd				1	0	1	sz	N	1	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
type = VFPNegMul_VNMUL;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP vectors These instructions can operate on VFP vectors under control of the FPSCR.LEN and FPSCR.STRIDE bits. For details see Appendix F *VFP Vector Operation Support*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.343 VNMLA、VNMLS、VNMUL

VNMLA は、2 つの浮動小数点レジスタの値を乗算し、その積の否定（符号反転）に、デスティネーションレジスタの浮動小数点数の否定（符号反転）を加算し、結果をデスティネーションレジスタに書き戻します。

VNMLS は、2 つの浮動小数点レジスタの値を乗算し、その積にデスティネーションレジスタの浮動小数点数の否定（符号反転）を加算し、結果をデスティネーションレジスタに書き戻します。

VNMUL は、2 つの浮動小数点レジスタの値を乗算し、その積の否定（符号反転）をデスティネーションレジスタに書き込みます。

エンコード T1/A1 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VNMLA<c>.F64 <Dd>, <Dn>, <Dm>
VNMLA<c>.F32 <Sd>, <Sn>, <Sm>
VNMLS<c>.F64 <Dd>, <Dn>, <Dm>
VNMLS<c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	0	1	Vn				Vd				1	0	1	sz	N	op	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	D	0	1	Vn				Vd				1	0	1	sz	N	op	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
type = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

エンコード T2/A2 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VNMUL<c>.F64 <Dd>, <Dn>, <Dm>
VNMUL<c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	1	0	Vn				Vd				1	0	1	sz	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	D	1	0	Vn				Vd				1	0	1	sz	N	1	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
type = VFPNegMul_VNMUL;
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP ベクタ これらの命令は、FPSCR.LEN ビットと FPSCR.STRIDE ビットの制御の下で VFP ベクタに作用することができます。詳細については、付録 F *「VFP ベクタ演算のサポート」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VN<op><c><q>.F64 <Dd>, <Dn>, <Dm>`	Encoding T1 / A1 with sz = 1
`VN<op><c><q>.F32 <Sd>, <Sn>, <Sm>`	Encoding T1 / A1 with sz = 0
`VNMUL<c><q>.F64 {<Dd>,} <Dn>, <Dm>`	Encoding T2 / A2 with sz = 1
`VNMUL<c><q>.F32 {<Sd>,} <Sn>, <Sm>`	Encoding T2 / A2 with sz = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<op>` Must be one of:

`MLA` op = 1

`MLS` op = 0.

`<Dd>, <Dn>, <Dm>` The destination register and the operand registers, for a double-precision operation.

`<Sd>, <Sn>, <Sm>` The destination register and the operand registers, for a single-precision operation.

Operation

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        product = FPMul(D[n], D[m], TRUE);
        case type of
            when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product), TRUE);
            when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product, TRUE);
            when VFPNegMul_VNMUL  D[d] = FPNeg(product);
    else
        product = FPMul(S[n], S[m], TRUE);
        case type of
            when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product), TRUE);
            when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product, TRUE);
            when VFPNegMul_VNMUL  S[d] = FPNeg(product);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact, Input Denormal.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`VN<op><c><q>.F64 <Dd>, <Dn>, <Dm>`	エンコード T1/A1、sz = 1
`VN<op><c><q>.F32 <Sd>, <Sn>, <Sm>`	エンコード T1/A1、sz = 0
`VNMUL<c><q>.F64 {<Dd>,} <Dn>, <Dm>`	エンコード T2/A2、sz =1
`VNMUL<c><q>.F32 {<Sd>,} <Sn>, <Sm>`	エンコード T2/A2、sz =0

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<op>` 次のいずれかを使用できます。

`MLA` op = 1

`MLS` op = 0

`<Dd>, <Dn>, <Dm>` 倍精度演算の場合のデスティネーションレジスタとオペランドレジスタ

`<Sd>, <Sn>, <Sm>` 単精度演算の場合のデスティネーションレジスタとオペランドレジスタ

動作

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        product = FPMul(D[n], D[m], TRUE);
        case type of
            when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product), TRUE);
            when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product, TRUE);
            when VFPNegMul_VNMUL  D[d] = FPNeg(product);
    else
        product = FPMul(S[n], S[m], TRUE);
        case type of
            when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product), TRUE);
            when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product, TRUE);
            when VFPNegMul_VNMUL  S[d] = FPNeg(product);
```

例外

未定義命令

浮動小数点例外：無効演算、オーバフロー、アンダーフロー、不正確、入力非正規化

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.344 VORN (immediate)

VORN (immediate) is a pseudo-instruction, equivalent to a VORR (immediate) instruction with the immediate value bitwise inverted. For details see *VORR (immediate)* on page A8-678.

A8.6.345 VORN (register)

This instruction performs a bitwise OR NOT operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Encoding T1 / A1 Advanced SIMD

```
VORN<c> <Qd>, <Qn>, <Qm>
VORN<c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.344 VORN（イミディエート）

VORN（イミディエート）は擬似命令で、イミディエート値がビット単位で反転された VORR（イミディエート）命令と等価です。詳細については、*VORR（イミディエート）* P. A8- 678 を参照して下さい。

A8.6.345 VORN（レジスタ）

VORN（レジスタ）は、2 つのレジスタの間でビット単位の否定論理和演算を実行し、結果をデスティネーションレジスタに配置します。オペランドレジスタと結果レジスタには、クワッドワードまたはダブルワードを使用できます。オペランドと結果は、すべて同じサイズの必要があります。

エンコード T1/A1 アドバンスト SIMD

```
VORN<c> <Qd>, <Qn>, <Qm>
VORN<c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VORN<c><q>{.<dt>}  {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VORN<c><q>{.<dt>}  {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM `VORN` instruction must be unconditional.
`<dt>`	An optional data type. It is ignored by assemblers, and does not affect the encoding.
`<Qd>, <Qn>, <Qm>`	The destination vector and the operand vectors, for a quadword operation.
`<Dd>, <Dn>, <Dm>`	The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR NOT(D[m+r]);
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VORN<c><q>{.<dt>}  {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VORN<c><q>{.<dt>}  {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VORN` は常に無条件命令です。
`<dt>`	オプションのデータ型。アセンブラによって無視され、エンコードに影響を与えません。
`<Qd>, <Qn>, <Qm>`	クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Dd>, <Dn>, <Dm>`	ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR NOT(D[m+r]);
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.346 VORR (immediate)

This instruction takes the contents of the destination vector, performs a bitwise OR with an immediate constant, and returns the result into the destination vector. For the range of constants available, see *One register and a modified immediate value* on page A7-21.

Encoding T1 / A1 Advanced SIMD

```
VORR<c>.<dt> <Qd>, #<imm>
VORR<c>.<dt> <Dd>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd				cmode				0	Q	0	1	imm4			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd				cmode				0	Q	0	1	imm4			

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE VMOV (immediate);
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.346 VORR（イミディエート）

VORR（イミディエート）は、デスティネーションベクタの内容を取得し、イミディエート定数とのビット単位の論理和を実行し、結果をデスティネーションベクタに返します。使用可能な定数の範囲については、*1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VORR<c>.<dt> <Qd>, #<imm>
VORR<c>.<dt> <Dd>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd				cmode				0	Q	0	1	imm4			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3			Vd				cmode				0	Q	0	1	imm4			

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE VMOV (immediate);
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VORR<c><q>.<dt>  {<Qd>,} <Qd>, #<imm>        Encoded as Q = 1
VORR<c><q>.<dt>  {<Dd>,} <Dd>, #<imm>>       Encoded as Q = 0
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7. An ARM VORR instruction must be unconditional.

<dt> The data type used for <imm>. It can be either I16 or I32.

I8, I64, and F32 are also permitted, but the resulting syntax is a pseudo-instruction.

<Qd> The destination vector for a quadword operation.

<Dd> The destination vector for a doubleword operation.

<imm> A constant of the type specified by <dt>. This constant is replicated enough times to fill the destination register. For example, VORR.I32 D0,#10 ORs 0x0000000A0000000A into D0.

For details of the range of constants available, and the encoding of <dt> and <imm>, see *One register and a modified immediate value* on page A7-21.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[d+r] OR imm64;
```

Exceptions

Undefined Instruction.

Pseudo-instructions

VORN can be used, with a range of constants that are the bitwise inverse of the available constants for VORR. This is assembled as the equivalent VORR instruction. Disassembly produces the VORR form.

One register and a modified immediate value on page A7-21 describes pseudo-instructions with a combination of <dt> and <imm> that is not supported by hardware, but that generates the same destination register value as a different combination that is supported by hardware.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VORR<c><q>.<dt>  {<Qd>,} <Qd>, #<imm>        Q = 1 としてエンコードされる
VORR<c><q>.<dt>  {<Dd>,} <Dd>, #<imm>>       Q = 0 としてエンコードされる
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM VORR は常に無条件命令です。

<dt> <imm> で使用されるデータ型。I16 または I32 を指定できます。

I8、I64、および F32 も指定できますが、指定した場合、構文が擬似命令になります。

<Qd> クワッドワード演算の場合のデスティネーションベクタ

<Dd> ダブルワード演算の場合のデスティネーションベクタ

<imm> <dt> で指定されたデータ型の定数。この定数は、デスティネーションレジスタを満たすのに必要な回数だけ複製されます。例えば、VORR.I32 D0,#10 では、0x0000000A0000000A の論理和が実行されて、D0 に書き込まれます。

使用可能な定数の範囲と、<dt> および <imm> のエンコードの詳細については、*1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[d+r] OR imm64;
```

例外

未定義命令

擬似命令

VORN は、VORR で使用可能な定数のビット単位の否定である定数の範囲で使用することができます。この命令は、等価な VORR 命令としてアセンブルされます。逆アセンブリでは、VORR 形式が生成されます。

ハードウェアでサポートされていないにもかかわらず、ハードウェアでサポートされている別の組み合わせと同じデスティネーションレジスタの値を生成する、<dt> と <imm> の組み合わせを持つ擬似命令については、*1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.347 VORR (register)

This instruction performs a bitwise OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Encoding T1 / A1 Advanced SIMD

```
VORR<c> <Qd>, <Qn>, <Qm>
VORR<c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

```
if N == M && Vn == Vm then SEE VMOV (register);
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.347 VORR（レジスタ）

VORR（レジスタ）命令は、2 つのレジスタの間でビット単位の論理和演算を実行し、結果をデスティネーションレジスタに配置します。オペランドレジスタと結果レジスタには、クワッドワードまたはダブルワードを使用できます。オペランドと結果は、すべて同じサイズの必要があります。

エンコード T1/A1 アドバンスト SIMD

```
VORR<c> <Qd>, <Qn>, <Qm>
VORR<c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

```
if N == M && Vn == Vm then SEE VMOV (register);
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VORR<c><q>{.<dt>}  {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VORR<c><q>{.<dt>}  {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM `VORR` instruction must be unconditional.
`<dt>`	An optional data type. It is ignored by assemblers, and does not affect the encoding.
`<Qd>, <Qn>, <Qm>`	The destination vector and the operand vectors, for a quadword operation.
`<Dd>, <Dn>, <Dm>`	The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR D[m+r];
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VORR<c><q>{.<dt>}  {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VORR<c><q>{.<dt>}  {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VORR` は常に無条件命令です。
`<dt>`	オプションのデータ型。アセンブラによって無視され、エンコードに影響を与えません。
`<Qd>, <Qn>, <Qm>`	クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Dd>, <Dn>, <Dm>`	ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR D[m+r];
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.348 VPADAL

Vector Pairwise Add and Accumulate Long adds adjacent pairs of elements of a vector, and accumulates the absolute values of the results into the elements of the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

Figure A8-2 shows an example of the operation of `VPADAL`.

Encoding T1 / A1 Advanced SIMD

```
VPADAL<c>.<dt>  <Qd>, <Qm>
VPADAL<c>.<dt>  <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	1	0	op	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	1	0	op	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Figure A8-2 Operation of doubleword VPADAL for data type S16

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.348 VPADAL

VPADAL（ベクタペア単位ロング加算および累積）は、ベクタの隣接するエレメントのペアを加算し、結果の絶対値をデスティネーションベクタのエレメントに累積します。

ベクタには、ダブルワードまたはクワッドワードを使用できます。オペランドのエレメントには、8 ビット、16 ビット、または 32 ビットの整数を使用できます。結果のエレメントの長さは、オペランドのエレメントの 2 倍です。

`VPADAL` の動作の例を、図 A8-2 に示します。

エンコード T1/A1 アドバンスト SIMD

```
VPADAL<c>.<dt>  <Qd>, <Qm>
VPADAL<c>.<dt>  <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	1	0	op	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	1	0	op	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

図 A8-2 データ型が S16 の場合のダブルワード VPADAL の動作

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VPADAL<c><q>.<dt> <Qd>, <Qm>`	Encoded as Q = 1
`VPADAL<c><q>.<dt> <Dd>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VPADAL` instruction must be unconditional.

`<dt>` The data type for the elements of the vectors. It must be one of:

- `S8` encoded as size = 0b00, op = 0
- `S16` encoded as size = 0b01, op = 0
- `S32` encoded as size = 0b10, op = 0
- `U8` encoded as size = 0b00, op = 1
- `U16` encoded as size = 0b01, op = 1
- `U32` encoded as size = 0b10, op = 1.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements/2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize];  op2 = Elem[D[m+r],2*e+1,esize];
            result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = Elem[D[d+r],e,2*esize] + result;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VPADAL<c><q>.<dt> <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VPADAL<c><q>.<dt> <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VPADAL` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S8` size = 0b00、op = 0 としてエンコードされます。
- `S16` size = 0b01、op = 0 としてエンコードされます。
- `S32` size = 0b10、op = 0 としてエンコードされます。
- `U8` size = 0b00、op = 1 としてエンコードされます。
- `U16` size = 0b01、op = 1 としてエンコードされます。
- `U32` size = 0b10、op = 1 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements/2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize];  op2 = Elem[D[m+r],2*e+1,esize];
            result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = Elem[D[d+r],e,2*esize] + result;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.349 VPADD (integer)

Vector Pairwise Add (integer) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.

The operand and result elements must all be the same type, and can be 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

Figure A8-3 shows an example of the operation of `VPADD`.

Encoding T1 / A1 Advanced SIMD

```
VPADD<c>.<dt>  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	1	Vm			

```
if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Figure A8-3 Operation of VPADD for data type `I16`

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.349 VPADD（整数）

VPADD（ベクタペア単位加算（整数））は、2 つのベクタの隣接するエレメントのペアを加算し、結果をデスティネーションベクタに配置します。

オペランドと結果はダブルワードベクタです。

オペランドのエレメントと結果のエレメントはすべて同じ型で、8 ビット、16 ビット、または 32 ビットの整数を使用できます。整数には符号付きと符号なしの区別はありません。

`VPADD` の動作の例を、図 A8-3 に示します。

エンコード T1/A1 アドバンスト SIMD

```
VPADD<c>.<dt>  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	1	Vm			

```
if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

図 A8-3 データ型が `I16` **の場合の VPADD の動作**

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VPADD<c><q>.<dt> {<Dd>,} <Dn>, <Dm>` Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VPADD` instruction must be unconditional.

`<dt>` The data type for the elements of the vectors. It must be one of:

- `I8` encoding T1 / A1, size = 0b00
- `I16` encoding T1 / A1, size = 0b01
- `I32` encoding T1 / A1, size = 0b10.

`<Dd>, <Dn>, <Dm>` The destination vector, the first operand vector, and the second operand vector.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements/2;

    for e = 0 to h-1
        Elem[dest,e,esize]   = Elem[D[n],2*e,esize] + Elem[D[n],2*e+1,esize];
        Elem[dest,e+h,esize] = Elem[D[m],2*e,esize] + Elem[D[m],2*e+1,esize];

    D[d] = dest;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VPADD<c><q>.<dt> {<Dd>,} <Dn>, <Dm>` Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VPADD` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `I8` エンコード T1/A1、size = 0b00
- `I16` エンコード T1/A1、size = 0b01
- `I32` エンコード T1/A1、size = 0b10

`<Dd>, <Dn>, <Dm>` デスティネーションベクタ、最初のオペランドベクタ、および 2 番目のオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements/2;

    for e = 0 to h-1
        Elem[dest,e,esize]   = Elem[D[n],2*e,esize] + Elem[D[n],2*e+1,esize];
        Elem[dest,e+h,esize] = Elem[D[m],2*e,esize] + Elem[D[m],2*e+1,esize];

    D[d] = dest;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.350 VPADD (floating-point)

Vector Pairwise Add (floating-point) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.

The operand and result elements are 32-bit floating-point numbers.

Figure A8-3 on page A8-684 shows an example of the operation of `VPADD`.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
VPADD<c>.F32  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

```
if sz == '1' || Q == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.350 VPADD（浮動小数点数）

VPADD（ベクタペア単位加算（浮動小数点数））は、2 つのベクタの隣接するエレメントのペアを加算し、結果をデスティネーションベクタに配置します。

オペランドと結果はダブルワードベクタです。

オペランドと結果のエレメントは、32 ビットの浮動小数点数です。

`VPADD` の動作の例を、P. A8-684 の図 A8-3 に示します。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VPADD<c>.F32  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

```
if sz == '1' || Q == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VPADD<c><q>.F32 {<Dd>,} <Dn>, <Dm>` Encoded as Q = 0, sz = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VPADD` instruction must be unconditional.

`<Dd>, <Dn>, <Dm>` The destination vector, the first operand vector, and the second operand vector.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements/2;

    for e = 0 to h-1
        Elem[dest,e,esize]   = FPAdd(Elem[D[n],2*e,esize], Elem[D[n],2*e+1,esize], FALSE);
        Elem[dest,e+h,esize] = FPAdd(Elem[D[m],2*e,esize], Elem[D[m],2*e+1,esize], FALSE);

    D[d] = dest;
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, Overflow, Underflow, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`VPADD<c><q>.F32 {<Dd>,} <Dn>, <Dm>` Q = 0、sz = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VPADD` は常に無条件命令です。

`<Dd>, <Dn>, <Dm>` デスティネーションベクタ、最初のオペランドベクタ、および 2 番目のオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements/2;

    for e = 0 to h-1
        Elem[dest,e,esize]   = FPAdd(Elem[D[n],2*e,esize], Elem[D[n],2*e+1,esize], FALSE);
        Elem[dest,e+h,esize] = FPAdd(Elem[D[m],2*e,esize], Elem[D[m],2*e+1,esize], FALSE);

    D[d] = dest;
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、オーバフロー、アンダーフロー、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.351 VPADDL

Vector Pairwise Add Long adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

Figure A8-4 shows an example of the operation of `VPADDL`.

Encoding T1 / A1 Advanced SIMD

```
VPADDL<c>.<dt>  <Qd>, <Qm>
VPADDL<c>.<dt>  <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	0	1	0	op	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	0	1	0	op	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Figure A8-4 Operation of doubleword VPADDL for data type S16

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.351 VPADDL

VPADDL（ベクタペア単位ロング加算）は、2 つのベクタの隣接するエレメントのペアを加算し、結果をデスティネーションベクタに配置します。

ベクタには、ダブルワードまたはクワッドワードを使用できます。オペランドのエレメントには、8 ビット、16 ビット、または 32 ビットの整数を使用できます。結果のエレメントの長さは、オペランドのエレメントの 2 倍です。

`VPADDL` の動作の例を、図 A8-4 に示します。

エンコード T1/A1 アドバンスト SIMD

```
VPADDL<c>.<dt>  <Qd>, <Qm>
VPADDL<c>.<dt>  <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	0	1	0	op	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	0	1	0	op	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

図 A8-4 データ型が S16 の場合のダブルワード VPADDL の動作

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VPADDL<c><q>.<dt>  <Qd>, <Qm>`	Encoded as Q = 1
`VPADDL<c><q>.<dt>  <Dd>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VPADDL` instruction must be unconditional.

`<dt>` The data type for the elements of the vectors. It must be one of:

`S8`	encoded as size = 0b00, op = 0
`S16`	encoded as size = 0b01, op = 0
`S32`	encoded as size = 0b10, op = 0
`U8`	encoded as size = 0b00, op = 1
`U16`	encoded as size = 0b01, op = 1
`U32`	encoded as size = 0b10, op = 1.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements/2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize];  op2 = Elem[D[m+r],2*e+1,esize];
            result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = result<2*esize-1:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VPADDL<c><q>.<dt>  <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VPADDL<c><q>.<dt>  <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VPADDL` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

`S8`	size = 0b00、op = 0 としてエンコードされます。
`S16`	size = 0b01、op = 0 としてエンコードされます。
`S32`	size = 0b10、op = 0 としてエンコードされます。
`U8`	size = 0b00、op = 1 としてエンコードされます。
`U16`	size = 0b01、op = 1 としてエンコードされます。
`U32`	size = 0b10、op = 1 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements/2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize];  op2 = Elem[D[m+r],2*e+1,esize];
            result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = result<2*esize-1:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.352 VPMAX, VPMIN (integer)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Figure A8-5 shows an example of the operation of `VPMAX`.

Encoding T1 / A1 Advanced SIMD

```
VP<op><c>.<dt>  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				1	0	1	0	N	Q	M	op	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				1	0	1	0	N	Q	M	op	Vm			

```
if size == '11' || Q == '1' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Figure A8-5 Operation of VPMAX for data type `S16` **or** `U16`

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.352 VPMAX、VPMIN（整数）

VPMAX（ベクタペア単位最大（整数））は、2 つのダブルワードベクタの隣接するエレメントのペアを比較し、大きい方のエレメントを、デスティネーション ダブルワードベクタの対応するエレメントにコピーします。

VPMIN（ベクタペア単位最小（整数））は、2 つのダブルワードベクタの隣接するエレメントのペアを比較し、小さい方のエレメントを、デスティネーション ダブルワードベクタの対応するエレメントにコピーします。

`VPMAX` の動作の例を、図 A8- 5 に示します。

エンコード T1/A1 アドバンスト SIMD

```
VP<op><c>.<dt>  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				1	0	1	0	N	Q	M	op	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				1	0	1	0	N	Q	M	op	Vm			

```
if size == '11' || Q == '1' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

図 A8- 5 データ型が S16 または U16 の場合の VPMAX の動作

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VP<op><c><q>.<dt> {<Dd>,} <Dn>, <Dm>` Encoded as Q = 0

where:

`<op>` Must be one of:
- `MAX` encoded as op = 0
- `MIN` encoded as op = 1.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VPMAX` or `VPMIN` instruction must be unconditional.

`<dt>` The data type for the elements of the vectors. It must be one of:
- `S8` encoding T1 / A1, size = 0b00, U = 0
- `S16` encoding T1 / A1, size = 0b01, U = 0
- `S32` encoding T1 / A1, size = 0b10, U = 0
- `U8` encoding T1 / A1, size = 0b00, U = 1
- `U16` encoding T1 / A1, size = 0b01, U = 1
- `U32` encoding T1 / A1, size = 0b10, U = 1.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements/2;

    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
        op1 = Int(Elem[D[m],2*e,esize], unsigned);
        op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e+h,esize] = result<esize-1:0>;

    D[d] = dest;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VP<op><c><q>.<dt> {<Dd>,} <Dn>, <Dm>` Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<op>` 次のいずれかを使用できます。
- `MAX` op = 0 としてエンコードされます。
- `MIN` op = 1 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VPMAX` または `VPMIN` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。
- `S8` エンコード T1/A1、size = 0b00、U = 0
- `S16` エンコード T1/A1、size = 0b01、U = 0
- `S32` エンコード T1/A1、size = 0b10、U = 0
- `U8` エンコード T1/A1、size = 0b00、U = 1
- `U16` エンコード T1/A1、size = 0b01、U = 1
- `U32` エンコード T1/A1、size = 0b10、U = 1

`<Dd>, <Dn>, <Dm>` デスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements/2;

    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
        op1 = Int(Elem[D[m],2*e,esize], unsigned);
        op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e+h,esize] = result<esize-1:0>;

    D[d] = dest;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.353 VPMAX, VPMIN (floating-point)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Figure A8-5 on page A8-690 shows an example of the operation of VPMAX.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
VP<op><c>.F32  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	op	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	op	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm			

```
if sz == '1' || Q == '1' then UNDEFINED;
maximum = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.353 VPMAX、VPMIN（浮動小数点数）

VPMAX（ベクタペア単位最大（浮動小数点数））は、2 つのダブルワードベクタの隣接するエレメントのペアを比較し、大きい方のエレメントを、デスティネーション ダブルワードベクタの対応するエレメントにコピーします。

VPMIN（ベクタペア単位最小（浮動小数点数））は、2 つのダブルワードベクタの隣接するエレメントのペアを比較し、小さい方のエレメントを、デスティネーション ダブルワードベクタの対応するエレメントにコピーします。

VPMAX の動作の例を、P. A8- 690 の図 A8- 5 に示します。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VP<op><c>.F32  <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	op	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	op	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm			

```
if sz == '1' || Q == '1' then UNDEFINED;
maximum = (op == '0');  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VP<op><c><q>.F32 {<Dd>,} <Dn>, <Dm>` Encoded as Q = 0, sz = 0

where:

`<op>` Must be one of:

- `MAX` encoded as op = 0
- `MIN` encoded as op = 1.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VPMAX` or `VPMIN` instruction must be unconditional.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements/2;

    for e = 0 to h-1
        op1 = Elem[D[n],2*e,esize];  op2 = Elem[D[n],2*e+1,esize];
        Elem[dest,e,esize] = if maximum then FPMax(op1,op2,FALSE) else FPMin(op1,op2,FALSE);
        op1 = Elem[D[m],2*e,esize];  op2 = Elem[D[m],2*e+1,esize];
        Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,FALSE) else FPMin(op1,op2,FALSE);

    D[d] = dest;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VP<op><c><q>.F32 {<Dd>,} <Dn>, <Dm>` Q = 0、sz = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<op>` 次のいずれかを使用できます。

- `MAX` op = 0 としてエンコードされます。
- `MIN` op = 1 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VPMAX` または `VPMIN` は常に無条件命令です。

`<Dd>, <Dn>, <Dm>` デスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements/2;

    for e = 0 to h-1
        op1 = Elem[D[n],2*e,esize];  op2 = Elem[D[n],2*e+1,esize];
        Elem[dest,e,esize] = if maximum then FPMax(op1,op2,FALSE) else FPMin(op1,op2,FALSE);
        op1 = Elem[D[m],2*e,esize];  op2 = Elem[D[m],2*e+1,esize];
        Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,FALSE) else FPMin(op1,op2,FALSE);

    D[d] = dest;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.354 VPOP

Vector Pop loads multiple consecutive extension registers from the stack.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD

`VPOP <list>` `<list>` is consecutive 64-bit registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	1	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	1	imm8							

```
single_regs = FALSE;  d = UInt(D:Vd);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FLDMX".
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
```

Encoding T2 / A2 VFPv2, VFPv3

`VPOP <list>` `<list>` is consecutive 32-bit registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	0	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	0	imm8							

```
single_regs = TRUE;  d = UInt(Vd:D);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
```

FLDMX Encoding T1/A1 behaves as described by the pseudocode if imm8 is odd. However, there is no UAL syntax for such encodings and their use is deprecated. For more information, see *FLDMX, FSTMX* on page A8-101.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.354 VPOP

VPOP（ベクタポップ）は、複数の連続する拡張機能レジスタをスタックからロードします。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD

`VPOP <list>` `<list>` は連続する 64 ビットレジスタ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	1	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	1	imm8							

```
single_regs = FALSE;  d = UInt(D:Vd);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FLDMX".
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
```

エンコード T2/A2 VFPv2、VFPv3

`VPOP <list>` `<list>` は連続する 32 ビットレジスタ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	0	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	0	imm8							

```
single_regs = TRUE;  d = UInt(Vd:D);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
```

FLDMX imm8 が奇数の場合、エンコード T1/A1 は擬似コードで記述されているように動作します。しかし、このようなエンコード用の UAL 構文は存在せず、このようなエンコードの使用は推奨されません。詳細については、*FLDMX*、*FSTMX* P. A8-101 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VPOP<c><q>{.<size>}  <list>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<size> An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers in <list>.

<list> The extension registers to be loaded, as a list of consecutively numbered doubleword (encoding T1 / A1) or singleword (encoding T2 / A2) registers, separated by commas and surrounded by brackets. It is encoded in the instruction by setting D and Vd to specify the first register in the list, and imm8 to twice the number of registers in the list (encoding T1 / A1) or the number of registers in the list (encoding T2 / A2). <list> must contain at least one register, and not more than sixteen.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);  NullCheckIfThumbEE(13);
    address = SP;
    SP = SP + imm32;
    if single_regs then
        for r = 0 to regs-1
            S[d+r] = MemA[address,4];  address = address+4;
    else
        for r = 0 to regs-1
            word1 = MemA[address,4];  word2 = MemA[address+4,4];  address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            D[d+r] = if BigEndian() then word1:word2 else word2:word1;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VPOP<c><q>{.<size>}  <list>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<size> オプションのデータサイズ指定子。指定する場合、<list> に含まれるレジスタのサイズ（32 ビットまたは 64 ビット）に等しいことが必要です。

<list> ロードされる拡張機能レジスタ。カンマで区切られ、角括弧で囲まれた、番号が連続しているダブルワード（エンコード T1/A1）レジスタまたはシングルワード（エンコード T2/A2）レジスタのリストとして示されます。リストの最初のレジスタを指定するために D および Vd を設定し、リストに含まれるレジスタの数の 2 倍（エンコード T1/A1）またはリストに含まれるレジスタの数（エンコード T2/A2）を指定するために imm8 を設定することで、命令にエンコードされます。<list> には、少なくとも 1 つのレジスタを含む必要がありますが、16 を超えるレジスタを含むことはできません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);  NullCheckIfThumbEE(13);
    address = SP;
    SP = SP + imm32;
    if single_regs then
        for r = 0 to regs-1
            S[d+r] = MemA[address,4];  address = address+4;
    else
        for r = 0 to regs-1
            word1 = MemA[address,4];  word2 = MemA[address+4,4];  address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            D[d+r] = if BigEndian() then word1:word2 else word2:word1;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.355 VPUSH

Vector Push stores multiple consecutive extension registers to the stack.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD

VPUSH<c> <list> — <list> is consecutive 64-bit registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	1	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	1	imm8							

```
single_regs = FALSE;  d = UInt(D:Vd);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FSTMX".
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
```

Encoding T2 / A2 VFPv2, VFPv3

VPUSH<c> <list> — <list> is consecutive 32-bit registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	0	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	0	imm8							

```
single_regs = TRUE;  d = UInt(Vd:D);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
```

FSTMX Encoding T1/A1 behaves as described by the pseudocode if imm8 is odd. However, there is no UAL syntax for such encodings and their use is deprecated. For more information, see *FLDMX, FSTMX* on page A8-101.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.355 VPUSH

VPUSH（ベクタプッシュ）は、複数の連続する拡張機能レジスタをスタックにストアします。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD

VPUSH<c> <list> — <list> は連続する 64 ビットレジスタ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	1	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	1	imm8							

```
single_regs = FALSE;  d = UInt(D:Vd);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FSTMX".
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
```

エンコード T2/A2 VFPv2、VFPv3

VPUSH<c> <list> — <list> は連続する 32 ビットレジスタ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	0	imm8							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	0	imm8							

```
single_regs = TRUE;  d = UInt(Vd:D);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
```

FSTMX imm8 が奇数の場合、エンコード T1/A1 は擬似コードで記述されているように動作します。しかし、このようなエンコード用の UAL 構文は存在せず、このようなエンコードの使用は推奨されません。詳細については、*FLDMX*、*FSTMX* P. A8-101 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VPUSH<c><q>{.<size>}  <list>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<size> An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers in <list>.

<list> The extension registers to be stored, as a list of consecutively numbered doubleword (encoding T1 / A1) or singleword (encoding T2 / A2) registers, separated by commas and surrounded by brackets. It is encoded in the instruction by setting D and Vd to specify the first register in the list, and imm8 to twice the number of registers in the list (encoding T1 / A1), or the number of registers in the list (encoding T2 / A2). <list> must contain at least one register, and not more than sixteen.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);  NullCheckIfThumbEE(13);
    address = SP - imm32;
    SP = SP - imm32;
    if single_regs then
        for r = 0 to regs-1
            MemA[address,4] = S[d+r];  address = address+4;
    else
        for r = 0 to regs-1
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian() then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian() then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VPUSH<c><q>{.<size>}  <list>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<size> オプションのデータサイズ指定子。指定する場合、<list> に含まれるレジスタのサイズ（32 ビットまたは 64 ビット）に等しいことが必要です。

<list> ストアされる拡張機能レジスタ。カンマで区切られ、角括弧で囲まれた、番号が連続しているダブルワード（エンコード T1/A1）レジスタまたはシングルワード（エンコード T2/A2）レジスタのリストとして示されます。リストの最初のレジスタを指定するために D および Vd を設定し、リストに含まれるレジスタの数の 2 倍（エンコード T1/A1）またはリストに含まれるレジスタの数（エンコード T2/A2）を指定するために imm8 を設定することで、命令にエンコードされます。<list> には、少なくとも 1 つのレジスタを含む必要がありますが、16 を超えるレジスタを含むことはできません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);  NullCheckIfThumbEE(13);
    address = SP - imm32;
    SP = SP - imm32;
    if single_regs then
        for r = 0 to regs-1
            MemA[address,4] = S[d+r];  address = address+4;
    else
        for r = 0 to regs-1
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian() then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian() then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.356 VQABS

Vector Saturating Absolute takes the absolute value of each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQABS<c>.<dt> <Qd>,<Qm>
VQABS<c>.<dt> <Dd>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	1	1	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	1	1	0	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.356 VQABS

VQABS（ベクタ絶対値飽和）は、ベクタの各エレメントの絶対値を取得し、結果をデスティネーションベクタに配置します。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2- 9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQABS<c>.<dt> <Qd>,<Qm>
VQABS<c>.<dt> <Dd>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	1	1	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	1	1	0	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VQABS<c><q>.<dt> <Qd>, <Qm>`	Encoded as Q = 1
`VQABS<c><q>.<dt> <Dd>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VQABS` instruction must be unconditional.

`<dt>` The data type for the elements of the vectors. It must be one of:

- `S8` encoded as size = 0b00
- `S16` encoded as size = 0b01
- `S32` encoded as size = 0b10.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Abs(SInt(Elem[D[m+r],e,esize]));
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VQABS<c><q>.<dt> <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VQABS<c><q>.<dt> <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VQABS` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S8` size = 0b00 としてエンコードされます。
- `S16` size = 0b01 としてエンコードされます。
- `S32` size = 0b10 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Abs(SInt(Elem[D[m+r],e,esize]));
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSCR.QC = '1';
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.357 VQADD

Vector Saturating Add adds the values of corresponding elements of two vectors, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQADD<c>.<dt> <Qd>,<Qn>,<Qm>
VQADD<c>.<dt> <Dd>,<Dn>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	0	0	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	0	0	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.357 VQADD

VQADD（ベクタ飽和加算）は、2 つのベクタの対応するエレメントの値を加算し、結果をデスティネーションベクタに配置します。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2- 9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQADD<c>.<dt> <Qd>,<Qn>,<Qm>
VQADD<c>.<dt> <Dd>,<Dn>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	0	0	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	0	0	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VQADD<c><q>.<type><size> {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VQADD<c><q>.<type><size> {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VQADD` instruction must be unconditional.

`<type>` The data type for the elements of the vectors. It must be one of:

- `S` signed, encoded as U = 0
- `U` unsigned, encoded as U = 1.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10
- `64` encoded as size = 0b11.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            sum = Int(Elem[D[n+r],e,esize], unsigned) + Int(Elem[D[m+r],e,esize], unsigned);
            (Elem[D[d+r],e,esize], sat) = SatQ(sum, esize, unsigned);
            if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VQADD<c><q>.<type><size> {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VQADD<c><q>.<type><size> {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VQADD` は常に無条件命令です。

`<type>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S` 符号付き、U = 0 としてエンコードされます。
- `U` 符号なし、U = 1 としてエンコードされます。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。
- `64` size = 0b11 としてエンコードされます。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            sum = Int(Elem[D[n+r],e,esize], unsigned) + Int(Elem[D[m+r],e,esize], unsigned);
            (Elem[D[d+r],e,esize], sat) = SatQ(sum, esize, unsigned);
            if sat then FPSCR.QC = '1';
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.358 VQDMLAL, VQDMLSL

Vector Saturating Doubling Multiply Accumulate Long multiplies corresponding elements in two doubleword vectors, doubles the products, and accumulates the results into the elements of a quadword vector.

Vector Saturating Doubling Multiply Subtract Long multiplies corresponding elements in two doubleword vectors, subtracts double the products from corresponding elements of a quadword vector, and places the results in the same quadword vector.

In both instructions, the second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page A7-9.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQD<op><c>.<dt> <Qd>,<Dn>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	size		Vn				Vd				1	0	op	1	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	size		Vn				Vd				1	0	op	1	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

Encoding T2 / A2 Advanced SIMD

```
VQD<op><c>.<dt> <Qd>,<Dn>,<Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	size		Vn				Vd				0	op	1	1	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	size		Vn				Vd				0	op	1	1	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.358 VQDMLAL、VQDMLSL

VQDMLAL（ベクタ飽和ロング積和演算 2 倍）は、2 つのダブルワードベクタの対応するエレメントを乗算し、その積を 2 倍にし、結果をクワッドワードベクタのエレメントに累積します。

VQDMLSL（ベクタ飽和ロング積減算 2 倍）は、2 つのダブルワードベクタの対応するエレメントを乗算し、その積の 2 倍をクワッドワードベクタの対応するエレメントから減算し、結果を同じクワッドワードベクタに配置します。

どちらの命令でも、2 番目のオペランドとしてベクタの代わりにスカラを使用できます。スカラの詳細については、*アドバンスト SIMD スカラ* P. A7-9 を参照して下さい。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2-9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQD<op><c>.<dt> <Qd>,<Dn>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	size		Vn				Vd				1	0	op	1	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	size		Vn				Vd				1	0	op	1	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

エンコード T2/A2 アドバンスト SIMD

```
VQD<op><c>.<dt> <Qd>,<Dn>,<Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	size		Vn				Vd				0	op	1	1	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	size		Vn				Vd				0	op	1	1	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VQD<op><c><q>.<dt>  <Qd>, <Dn>, <Dm>
VQD<op><c><q>.<dt>  <Qd>, <Dn>, <Dm[x]>
```

where:

<op> Must be one of:
- MLAL encoded as op = 0
- MLSL encoded as op = 1.

<c><q> See *Standard assembler syntax fields* on page A8-7. An ARM VQDMLAL or VQDMLSL instruction must be unconditional.

<dt> The data type for the elements of the operands. It must be one of:
- S16 encoded as size = 0b01
- S32 encoded as size = 0b10.

<Qd>, <Dn> The destination vector and the first operand vector.

<Dm> The second operand vector, for an all vector operation.

<Dm[x]> The scalar for a scalar operation. If <dt> is S16, Dm is restricted to D0-D7. If <dt> is S32, Dm is restricted to D0-D15.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[D[m],e,esize]);
        op1 = SInt(Elem[D[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        if add then
            result = SInt(Elem[Q[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Q[d>>1],e,2*esize]) - SInt(product);
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VQD<op><c><q>.<dt>  <Qd>, <Dn>, <Dm>
VQD<op><c><q>.<dt>  <Qd>, <Dn>, <Dm[x]>
```

各項目の説明については以下を参照して下さい。

<op> 次のいずれかを使用できます。
- MLAL op = 0 としてエンコードされます。
- MLSL op = 1 としてエンコードされます。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM VQDMLAL および VQDMLSL は常に無条件命令です。

<dt> オペランドのエレメントのデータ型。次のいずれかを使用できます。
- S16 size = 0b01 としてエンコードされます。
- S32 size = 0b10 としてエンコードされます。

<Qd>, <Dn> デスティネーションベクタと最初のオペランドベクタ

<Dm> すべてベクタの演算の場合、2 番目のオペランドベクタ

<Dm[x]> スカラ演算の場合のスカラ。<dt> が S16 の場合、Dm は D0 ～ D7 に限定されます。<dt> が S32 の場合、Dm は D0 ～ D15 に限定されます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[D[m],e,esize]);
        op1 = SInt(Elem[D[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        if add then
            result = SInt(Elem[Q[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Q[d>>1],e,2*esize]) - SInt(product);
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSCR.QC = '1';
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.359 VQDMULH

Vector Saturating Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are truncated (for rounded results see *VQRDMULH* on page A8-712).

The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page A7-9.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQDMULH<c>.<dt> <Qd>,<Qn>,<Qm>
VQDMULH<c>.<dt> <Dd>,<Dn>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 Advanced SIMD

```
VQDMULH<c>.<dt> <Qd>,<Qn>,<Dm[x]>
VQDMULH<c>.<dt> <Dd>,<Dn>,<Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	size		Vn				Vd				1	1	0	0	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	size		Vn				Vd				1	1	0	0	N	1	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.359 VQDMULH

VQDMULH（上位半分を返すベクタ飽和乗算 2 倍）は、2 つのベクタの対応するエレメントを乗算し、結果を 2 倍にし、最終結果の上位半分をデスティネーションベクタに配置します。結果は切り捨てられます（丸められる結果については、*VQRDMULH* P. A8-712 を参照）。

2 番目のオペランドには、ベクタの代わりにスカラを使用できます。スカラの詳細については、*アドバンスト SIMD スカラ* P. A7-9 を参照して下さい。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2-9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQDMULH<c>.<dt> <Qd>,<Qn>,<Qm>
VQDMULH<c>.<dt> <Dd>,<Dn>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 アドバンスト SIMD

```
VQDMULH<c>.<dt> <Qd>,<Qn>,<Dm[x]>
VQDMULH<c>.<dt> <Dd>,<Dn>,<Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	size		Vn				Vd				1	1	0	0	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	size		Vn				Vd				1	1	0	0	N	1	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VQDMULH<c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Encoding T1 / A1, Q = 1
`VQDMULH<c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Encoding T1 / A1, Q = 0
`VQDMULH<c><q>.<dt> {<Qd>,} <Qn>, <Dm[x]>`	Encoding T2 / A2, U = 1
`VQDMULH<c><q>.<dt> {<Dd>,} <Dn>, <Dm[x]>`	Encoding T2 / A2, U = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VQDMULH` instruction must be unconditional.

`<dt>` The data type for the elements of the operands. It must be one of:

- `S16` encoded as size = 0b01
- `S32` encoded as size = 0b10.

`<Qd>, <Qn>` The destination vector and the first operand vector, for a quadword operation.

`<Dd>, <Dn>` The destination vector and the first operand vector, for a doubleword operation.

`<Qm>` The second operand vector, for a quadword all vector operation.

`<Dm>` The second operand vector, for a doubleword all vector operation.

`<Dm[x]>` The scalar for either a quadword or a doubleword scalar operation. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. If `<dt>` is `S32`, `Dm` is restricted to D0-D15.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            op1 = SInt(Elem[D[n+r],e,esize]);
            // The following only saturates if both op1 and op2 equal -(2^(esize-1))
            (result, sat) = SignedSatQ((2*op1*op2) >> esize, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VQDMULH<c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	エンコード T1/A1、Q = 1
`VQDMULH<c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	エンコード T1/A1、Q = 0
`VQDMULH<c><q>.<dt> {<Qd>,} <Qn>, <Dm[x]>`	エンコード T2/A2、U = 1
`VQDMULH<c><q>.<dt> {<Dd>,} <Dn>, <Dm[x]>`	エンコード T2/A2、U = 0

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VQDMULH` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S16` size = 0b01 としてエンコードされます。
- `S32` size = 0b10 としてエンコードされます。

`<Qd>, <Qn>` クワッドワード演算の場合のデスティネーションベクタと最初のオペランドベクタ

`<Dd>, <Dn>` ダブルワード演算の場合のデスティネーションベクタと最初のオペランドベクタ

`<Qm>` すべてクワッドワードベクタを使用する演算での、2 番目のオペランドベクタ

`<Dm>` すべてダブルワードのベクタを使用する演算での、2 番目のオペランドベクタ

`<Dm[x]>` クワッドワードまたはダブルワードのスカラ演算で使用するスカラ。`<dt>` が `S16` の場合、`Dm` は D0 ～ D7 に限定されます。`<dt>` が `S32` の場合、`Dm` は D0 ～ D15 に限定されます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            op1 = SInt(Elem[D[n+r],e,esize]);
            // The following only saturates if both op1 and op2 equal -(2^(esize-1))
            (result, sat) = SignedSatQ((2*op1*op2) >> esize, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.360 VQDMULL

Vector Saturating Doubling Multiply Long multiplies corresponding elements in two doubleword vectors, doubles the products, and places the results in a quadword vector.

The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page A7-9.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQDMULL<c>.<dt> <Qd>,<Dn>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	size		Vn				Vd				1	1	0	1	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	size		Vn				Vd				1	1	0	1	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

Encoding T2 / A2 Advanced SIMD

```
VQDMULL<c>.<dt> <Qd>,<Dn>,<Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	size		Vn				Vd				1	0	1	1	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	size		Vn				Vd				1	0	1	1	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.360 VQDMULL

VQDMULL（ベクタ飽和ロング乗算 2 倍）は、2 つのダブルワードベクタの対応するエレメントを乗算し、その積を 2 倍にし、結果をクワッドワードベクタに配置します。

2 番目のオペランドには、ベクタの代わりにスカラを使用できます。スカラの詳細については、*アドバンスト SIMD スカラ* P. A7-9 を参照して下さい。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2-9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQDMULL<c>.<dt> <Qd>,<Dn>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	size		Vn				Vd				1	1	0	1	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	size		Vn				Vd				1	1	0	1	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

エンコード T2/A2 アドバンスト SIMD

```
VQDMULL<c>.<dt> <Qd>,<Dn>,<Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	size		Vn				Vd				1	0	1	1	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	size		Vn				Vd				1	0	1	1	N	1	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VQDMULL<c><q>.<dt>  <Qd>, <Dn>, <Dm>
VQDMULL<c><q>.<dt>  <Qd>, <Dn>, <Dm[x]>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VQDMULL` instruction must be unconditional.

`<dt>` The data type for the elements of the operands. It must be one of:

- `S16` encoded as size = 0b01
- `S32` encoded as size = 0b10.

`<Qd>, <Dn>` The destination vector and the first operand vector.

`<Dm>` The second operand vector, for an all vector operation.

`<Dm[x]>` The scalar for a scalar operation. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. If `<dt>` is `S32`, `Dm` is restricted to D0-D15.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[D[m],e,esize]);
        op1 = SInt(Elem[D[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat) = SignedSatQ(2*op1*op2, 2*esize);
        Elem[Q[d>>1],e,2*esize] = product;
        if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VQDMULL<c><q>.<dt>  <Qd>, <Dn>, <Dm>
VQDMULL<c><q>.<dt>  <Qd>, <Dn>, <Dm[x]>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。ARM `VQDMULL` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S16` size = 0b01 としてエンコードされます。
- `S32` size = 0b10 としてエンコードされます。

`<Qd>, <Dn>` デスティネーションベクタと最初のオペランドベクタ

`<Dm>` すべてベクタの演算の場合、2 番目のオペランドベクタ

`<Dm[x]>` スカラ演算の場合のスカラ。`<dt>` が `S16` の場合、`Dm` は D0 ～ D7 に限定されます。`<dt>` が `S32` の場合、`Dm` は D0 ～ D15 に限定されます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[D[m],e,esize]);
        op1 = SInt(Elem[D[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat) = SignedSatQ(2*op1*op2, 2*esize);
        Elem[Q[d>>1],e,2*esize] = product;
        if sat then FPSCR.QC = '1';
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.361 VQMOVN, VQMOVUN

Vector Saturating Move and Narrow copies each element of the operand vector to the corresponding element of the destination vector.

The operand is a quadword vector. The elements can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result is a doubleword vector. The elements are half the length of the operand vector elements. If the operand is unsigned, the results are unsigned. If the operand is signed, the results can be signed or unsigned.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQMOV{U}N<c>.<type><size> <Dd>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	1	0	op		M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	1	0	op		M	0	Vm			

```
if op == '00' then SEE VMOVN;
if size == '11' || Vm<0> == '1' then UNDEFINED;
source_unsigned = (op == '11');  dest_unsigned = (op<0> == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.361 VQMOVN、VQMOVUN

VQMOVN、VQMOVUN（ベクタ飽和移動および縮小）は、オペランドベクタの各エレメントをデスティネーションベクタの対応するエレメントにコピーします。

オペランドはクワッドワードベクタです。エレメントには、次のいずれかを使用できます。

- 16 ビット、32 ビット、または 64 ビットの符号付き整数
- 16 ビット、32 ビット、または 64 ビットの符号なし整数

結果はダブルワードベクタです。結果のエレメントの長さは、オペランドベクタのエレメントの半分です。オペランドが符号なしの場合、結果も符号なしです。オペランドが符号付きの場合、結果は符号付きまたは符号なしです。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2- 9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQMOV{U}N<c>.<type><size> <Dd>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	1	0	op		M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	1	0	op		M	0	Vm			

```
if op == '00' then SEE VMOVN;
if size == '11' || Vm<0> == '1' then UNDEFINED;
source_unsigned = (op == '11');  dest_unsigned = (op<0> == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VQMOV{U}N<c><q>.<type><size>  <Dd>, <Qm>
```

where:

`U` If present, specifies that the operation produces unsigned results, even though the operands are signed. Encoded as op = 0b01.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VQMOVN` or `VQMOVUN` instruction must be unconditional.

`<type>` The data type for the elements of the operand. It must be one of:

- `S` encoded as:
  - op = 0b10 for `VQMOVN`
  - op = 0b01 for `VQMOVUN`.
- `U` encoded as op = 0b11. Not available for `VQMOVUN`.

`<size>` The data size for the elements of the operand. It must be one of:

- `16` encoded as size = 0b00
- `32` encoded as size = 0b01
- `64` encoded as size = 0b10.

`<Dd>, <Qm>` The destination vector and the operand vector.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        operand = Int(Elem[Q[m>>1],e,2*esize], src_unsigned);
        (Elem[D[d],e,esize], sat) = SatQ(operand, esize, dest_unsigned);
        if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VQMOV{U}N<c><q>.<type><size>  <Dd>, <Qm>
```

各項目の説明については以下を参照して下さい。

`U` 存在する場合、オペランドが符号付きであっても演算の結果を符号なしにすることが指定されます。op = 0b01 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VQMOVN` および `VQMOVUN` は常に無条件命令です。

`<type>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S` 次のようにエンコードされます。
  - `VQMOVN` では op = 0b10
  - `VQMOVUN` では op = 0b01
- `U` op = 0b11 としてエンコードされます。`VQMOVUN` では使用できません。

`<size>` オペランドのエレメントのデータサイズ。次のいずれかを使用できます。

- `16` size = 0b00 としてエンコードされます。
- `32` size = 0b01 としてエンコードされます。
- `64` size = 0b10 としてエンコードされます。

`<Dd>, <Qm>` デスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        operand = Int(Elem[Q[m>>1],e,2*esize], src_unsigned);
        (Elem[D[d],e,esize], sat) = SatQ(operand, esize, dest_unsigned);
        if sat then FPSCR.QC = '1';
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.362 VQNEG

Vector Saturating Negate negates each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQNEG<c>.<dt> <Qd>,<Qm>
VQNEG<c>.<dt> <Dd>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	1	1	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	1	1	1	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.362 VQNEG

VQNEG（ベクタ飽和否定）は、ベクタの各エレメントを反転し、結果をデスティネーションベクタに配置します。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2- 9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQNEG<c>.<dt> <Qd>,<Qm>
VQNEG<c>.<dt> <Dd>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	1	1	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	1	1	1	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VQNEG<c><q>.<dt> <Qd>, <Qm>`	Encoded as Q = 1
`VQNEG<c><q>.<dt> <Dd>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VQNEG` instruction must be unconditional.

`<dt>` The data type for the elements of the vectors. It must be one of:

- `S8` encoded as size = 0b00
- `S16` encoded as size = 0b01
- `S32` encoded as size = 0b10.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = -SInt(Elem[D[m+r],e,esize]);
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VQNEG<c><q>.<dt> <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VQNEG<c><q>.<dt> <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VQNEG` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S8` size = 0b00 としてエンコードされます。
- `S16` size = 0b01 としてエンコードされます。
- `S32` size = 0b10 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = -SInt(Elem[D[m+r],e,esize]);
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSCR.QC = '1';
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.363 VQRDMULH

Vector Saturating Rounding Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are rounded (for truncated results see *VQDMULH* on page A8-704).

The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars* on page A7-9.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQRDMULH<c>.<dt> <Qd>,<Qn>,<Qm>
VQRDMULH<c>.<dt> <Dd>,<Dn>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 Advanced SIMD

```
VQRDMULH<c>.<dt> <Qd>,<Qn>,<Dm[x]>
VQRDMULH<c>.<dt> <Dd>,<Dn>,<Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	size		Vn				Vd				1	1	0	1	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	size		Vn				Vd				1	1	0	1	N	1	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.363 VQRDMULH

VQRDMULH（上位半分を返すベクタ飽和丸め付き乗算 2 倍）は、2 つのベクタの対応するエレメントを乗算し、結果を 2 倍にし、最終結果の上位半分をデスティネーションベクタに配置します。結果は丸められます（切り捨てられる結果については、*VQDMULH* P. A8-704 を参照）。

2 番目のオペランドには、ベクタの代わりにスカラを使用できます。スカラの詳細については、*アドバンスト SIMD スカラ* P. A7-9 を参照して下さい。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2-9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQRDMULH<c>.<dt> <Qd>,<Qn>,<Qm>
VQRDMULH<c>.<dt> <Dd>,<Dn>,<Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 アドバンスト SIMD

```
VQRDMULH<c>.<dt> <Qd>,<Qn>,<Dm[x]>
VQRDMULH<c>.<dt> <Dd>,<Dn>,<Dm[x]>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	size		Vn				Vd				1	1	0	1	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	size		Vn				Vd				1	1	0	1	N	1	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VQRDMULH<c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Encoding T1 / A1, Q = 1
`VQRDMULH<c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Encoding T1 / A1, Q = 0
`VQRDMULH<c><q>.<dt> {<Qd>,} <Qn>, <Dm[x]>`	Encoding T2 / A2, Q = 1
`VQRDMULH<c><q>.<dt> {<Dd>,} <Dn>, <Dm[x]>`	Encoding T2 / A2, Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VQRDMULH` instruction must be unconditional.

`<dt>` The data type for the elements of the operands. It must be one of:

- `S16` encoded as size = 0b01
- `S32` encoded as size = 0b10.

`<Qd>, <Qn>` The destination vector and the first operand vector, for a quadword operation.

`<Dd>, <Dn>` The destination vector and the first operand vector, for a doubleword operation.

`<Qm>` The second operand vector, for a quadword all vector operation.

`<Dm>` The second operand vector, for a doubleword all vector operation.

`<Dm[x]>` The scalar for either a quadword or a doubleword scalar operation. If `<dt>` is `S16`, `Dm` is restricted to D0-D7. If `<dt>` is `S32`, `Dm` is restricted to D0-D15.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = SInt(Elem[D[n+r],e,esize]);
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            (result, sat) = SignedSatQ((2*op1*op2 + round_const) >> esize, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VQRDMULH<c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	エンコード T1/A1、Q = 1
`VQRDMULH<c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	エンコード T1/A1、Q = 0
`VQRDMULH<c><q>.<dt> {<Qd>,} <Qn>, <Dm[x]>`	エンコード T2/A2、Q = 1
`VQRDMULH<c><q>.<dt> {<Dd>,} <Dn>, <Dm[x]>`	エンコード T2/A2、Q = 0

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VQRDMULH` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S16` size = 0b01 としてエンコードされます。
- `S32` size = 0b10 としてエンコードされます。

`<Qd>, <Qn>` クワッドワード演算の場合のデスティネーションベクタと最初のオペランドベクタ

`<Dd>, <Dn>` ダブルワード演算の場合のデスティネーションベクタと最初のオペランドベクタ

`<Qm>` すべてクワッドワードベクタを使用する演算での、2 番目のオペランドベクタ

`<Dm>` すべてダブルワードのベクタを使用する演算での、2 番目のオペランドベクタ

`<Dm[x]>` クワッドワードまたはダブルワードのスカラ演算で使用するスカラ。`<dt>` が `S16` の場合、`Dm` は D0 ～ D7 に限定されます。`<dt>` が `S32` の場合、`Dm` は D0 ～ D15 に限定されます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = SInt(Elem[D[n+r],e,esize]);
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            (result, sat) = SignedSatQ((2*op1*op2 + round_const) >> esize, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.364 VQRSHL

Vector Saturating Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

For truncated results see *VQSHL (register)* on page A8-718.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQRSHL<c>.<type><size> <Qd>,<Qm>,<Qn>
VQRSHL<c>.<type><size> <Dd>,<Dm>,<Dn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.364 VQRSHL

VQRSHL（ベクタ飽和丸め付き左シフト）は、ベクタの各エレメントを取得して、2 番目のベクタの対応するエレメントの最下位バイトにより指定される値だけシフトし、結果をデスティネーションベクタに配置します。シフトの値が正の場合、演算は左シフトになります。それ以外の場合、右シフトになります。

切り捨てられる結果については、*VQSHL（レジスタ）* P. A8- 718 を参照して下さい。

最初のオペランドと結果のエレメントのデータ型は同じで、次のいずれかを使用できます。

- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号付き整数
- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号なし整数

2 番目のオペランドは、同じサイズの符号付き整数です。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2- 9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQRSHL<c>.<type><size> <Qd>,<Qm>,<Qn>
VQRSHL<c>.<type><size> <Dd>,<Dm>,<Dn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	0	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	0	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VQRSHL<c><q>.<type><size> {<Qd>,} <Qm>, <Qn>`	Encoded as Q = 1
`VQRSHL<c><q>.<type><size> {<Dd>,} <Dm>, <Dn>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VQRSHL` instruction must be unconditional.

`<type>` The data type for the elements of the vectors. It must be one of:

- `S` signed, encoded as U = 0
- `U` unsigned, encoded as U = 1.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10
- `64` encoded as size = 0b11.

`<Qd>, <Qm>, <Qn>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dm>, <Dn>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            round_const = 1 << (-1-shift); // 0 for left shift, 2^(n-1) for right shift
            operand = Int(Elem[D[m+r],e,esize], unsigned);
            (result, sat) = SatQ((operand + round_const) << shift, esize, unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VQRSHL<c><q>.<type><size> {<Qd>,} <Qm>, <Qn>`	Q = 1 としてエンコードされる
`VQRSHL<c><q>.<type><size> {<Dd>,} <Dm>, <Dn>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VQRSHL` は常に無条件命令です。

`<type>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S` 符号付き、U = 0 としてエンコードされます。
- `U` 符号なし、U = 1 としてエンコードされます。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。
- `64` size = 0b11 としてエンコードされます。

`<Qd>, <Qm>, <Qn>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>, <Dn>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            round_const = 1 << (-1-shift); // 0 for left shift, 2^(n-1) for right shift
            operand = Int(Elem[D[m+r],e,esize], unsigned);
            (result, sat) = SatQ((operand + round_const) << shift, esize, unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.365 VQRSHRN, VQRSHRUN

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the rounded results in a doubleword vector.

For truncated results, see *VQSHRN, VQSHRUN* on page A8-722.

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQRSHR{U}N<c>.<type><size> <Dd>,<Qm>,#<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				1	0	0	op	0	1	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				1	0	0	op	0	1	M	1	Vm			

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE VRSHRN;
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.365 VQRSHRN、VQRSHRUN

VQRSHRN、VQRSHRUN（ベクタ飽和丸め付き右シフトおよび縮小）は、整数のクワッドワードベクタの各エレメントを取得して、イミディエート値で指定されるビット数だけ右にシフトし、丸められた結果をダブルワードベクタに配置します。

切り捨てられる結果については、*VQSHRN、VQSHRUN* P. A8-722 を参照して下さい。

オペランドのエレメントはすべて同じサイズで、次のいずれかを使用できます。

- 16 ビット、32 ビット、または 64 ビットの符号付き整数
- 16 ビット、32 ビット、または 64 ビットの符号なし整数

結果のエレメントの幅は、オペランドのエレメントの半分です。オペランドのエレメントが符号付きの場合、結果のエレメントは符号付きまたは符号なしです。オペランドのエレメントが符号なしの場合、結果のエレメントも符号なしです。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2-9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQRSHR{U}N<c>.<type><size> <Dd>,<Qm>,#<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				1	0	0	op	0	1	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				1	0	0	op	0	1	M	1	Vm			

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE VRSHRN;
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VQRSHR{U}N<c><q>.<type><size>  <Dd>, <Qm>, #<imm>
```

where:

U — If present, specifies that the results are unsigned, although the operands are signed.

<c><q> — See *Standard assembler syntax fields* on page A8-7. An ARM VQRSHRN or VQRSHRUN instruction must be unconditional.

<type> — The data type for the elements of the vectors. It must be one of:

- S — encoded as:
  - U = 0, op = 1, if U is absent
  - U = 1, op = 0, if U is present
- U — encoded as U = 1, op = 1. Not available for VQRSHRUN.

<size> — The data size for the elements of the vectors. It must be one of:

- 16 — Encoded as L = '0', imm6<5:3> = '001'. (8– <imm>) is encoded in imm6<2:0>.
- 32 — Encoded as L = '0', imm6<5:4> = '01'. (16 – <imm>) is encoded in imm6<3:0>.
- 64 — Encoded as L = '0', imm6<5> = '1'. (32 – <imm>) is encoded in imm6<4:0>.

<Dd>, <Qm> — The destination vector and the operand vector.

<imm> — The immediate value, in the range 1 to <size>/2. See the description of <size> for how <imm> is encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for e = 0 to elements-1
        operand = Int(Elem[Q[m>>1],e,2*esize], src_unsigned);
        (result, sat) = SatQ((operand + round_const) >> shift_amount, esize, dest_unsigned);
        Elem[D[d],e,esize] = result;
        if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

Pseudo-instructions

VQRSHRN.I<size> <Dd>, <Qm>, #0	is a synonym for	VQMOVN.I<size> <Dd>, <Qm>
VQRSHRUN.I<size> <Dd>, <Qm>, #0	is a synonym for	VQMOVUN.I<size> <Dd>, <Qm>

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VQRSHR{U}N<c><q>.<type><size>  <Dd>, <Qm>, #<imm>
```

各項目の説明については以下を参照して下さい。

U — 存在する場合、オペランドが符号付きであっても結果を符号なしにすることが指定されます。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM VQRSHRN および VQRSHRUN は常に無条件命令です。

<type> — ベクタのエレメントのデータ型。次のいずれかを使用できます。

- S — 次のようにエンコードされます。
  - U = 0、op = 1（U を省略した場合）
  - U = 1、op = 0（U を指定した場合）
- U — U = 1、op = 1 としてエンコードされます。VQRSHRUN では使用できません。

<size> — ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- 16 — L = '0'、imm6〈5:3〉= '001' としてエンコードされます。(8 - <imm>) は、imm6〈2:0〉にエンコードされます。
- 32 — L = '0'、imm6〈5:4〉= '01' としてエンコードされます。(16 - <imm>) は、imm6〈3:0〉にエンコードされます。
- 64 — L = '0'、imm6〈5〉= '1' としてエンコードされます。(32 - <imm>) は、imm6〈4:0〉にエンコードされます。

<Dd>, <Qm> — デスティネーションベクタとオペランドベクタ

<imm> — 1 ～ <size> / 2 の範囲のイミディエート値。<imm> のエンコード方法については、<size> の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for e = 0 to elements-1
        operand = Int(Elem[Q[m>>1],e,2*esize], src_unsigned);
        (result, sat) = SatQ((operand + round_const) >> shift_amount, esize, dest_unsigned);
        Elem[D[d],e,esize] = result;
        if sat then FPSCR.QC = '1';
```

例外

未定義命令

擬似命令

VQRSHRN.I<size> <Dd>, <Qm>, #0	は次の命令と同義語	VQMOVN.I<size> <Dd>, <Qm>
VQRSHRUN.I<size> <Dd>, <Qm>, #0	は次の命令と同義語	VQMOVUN.I<size> <Dd>, <Qm>

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.366 VQSHL (register)

Vector Saturating Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

The results are truncated. For rounded results, see *VQRSHL* on page A8-714.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQSHL<c>.<type><size> <Qd>,<Qm>,<Qn>
VQSHL<c>.<type><size> <Dd>,<Dm>,<Dn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	0	0	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	0	0	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.366 VQSHL（レジスタ）

VQSHL（ベクタ飽和左シフト（レジスタ））は、ベクタの各エレメントを取得して、2 番目のベクタの対応するエレメントの最下位バイトにより指定されるビット数だけシフトし、結果をデスティネーションベクタに配置します。シフトの値が正の場合、演算は左シフトになります。それ以外の場合、右シフトになります。

結果は切り捨てられます。丸められる結果については、*VQRSHL* P. A8- 714 を参照して下さい。

最初のオペランドと結果のエレメントのデータ型は同じで、次のいずれかを使用できます。

- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号付き整数
- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号なし整数

2 番目のオペランドは、同じサイズの符号付き整数です。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2- 9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQSHL<c>.<type><size> <Qd>,<Qm>,<Qn>
VQSHL<c>.<type><size> <Dd>,<Dm>,<Dn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	0	0	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	0	0	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VQSHL<c><q>.<type><size> {<Qd>,} <Qm>, <Qn>`	Encoded as Q = 1
`VQSHL<c><q>.<type><size> {<Dd>,} <Dm>, <Dn>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VQSHL` instruction must be unconditional.

`<type>` The data type for the elements of the vectors. It must be one of:

- `S` signed, encoded as U = 0
- `U` unsigned, encoded as U = 1.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10
- `64` encoded as size = 0b11.

`<Qd>, <Qm>, <Qn>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dm>, <Dn>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            operand = Int(Elem[D[m+r],e,esize], unsigned);
            (result,sat) = SatQ(operand << shift, esize, unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VQSHL<c><q>.<type><size> {<Qd>,} <Qm>, <Qn>`	Q = 1 としてエンコードされる
`VQSHL<c><q>.<type><size> {<Dd>,} <Dm>, <Dn>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VQSHL` は常に無条件命令です。

`<type>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S` 符号付き、U = 0 としてエンコードされます。
- `U` 符号なし、U = 1 としてエンコードされます。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。
- `64` size = 0b11 としてエンコードされます。

`<Qd>, <Qm>, <Qn>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>, <Dn>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            operand = Int(Elem[D[m+r],e,esize], unsigned);
            (result,sat) = SatQ(operand << shift, esize, unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.367 VQSHL, VQSHLU (immediate)

Vector Saturating Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in a second vector.

The operand elements must all be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are the same size as the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQSHL{U}<c>.<type><size> <Qd>,<Qm>,#<imm>
VQSHL{U}<c>.<type><size> <Dd>,<Dm>,#<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	1	1	op	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	1	1	op	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.367 VQSHL、VQSHLU（イミディエート）

VQSHL、VQSHLU（ベクタ飽和左シフト（イミディエート））は、整数のベクタの各エレメントを取得して、イミディエート値分だけ左にシフトし、結果を 2 番目のベクタに配置します。

オペランドのエレメントはすべて同じサイズで、次のいずれかを使用できます。

- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号付き整数
- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号なし整数

結果のエレメントのサイズは、オペランドのエレメントと同じです。オペランドのエレメントが符号付きの場合、結果のエレメントは符号付きまたは符号なしです。オペランドのエレメントが符号なしの場合、結果のエレメントも符号なしです。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2-9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQSHL{U}<c>.<type><size> <Qd>,<Qm>,#<imm>
VQSHL{U}<c>.<type><size> <Dd>,<Dm>,#<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	1	1	op	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	1	1	op	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VQSHL{U}<c><q>.<type><size>  {<Qd>,} <Qm>, #<imm>        Encoded as Q = 1
VQSHL{U}<c><q>.<type><size>  {<Dd>,} <Dm>, #<imm>        Encoded as Q = 0
```

where:

`U` If present, specifies that the results are unsigned, although the operands are signed.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VQSHL` or `VQSHLU` instruction must be unconditional.

`<type>` The data type for the elements of the vectors. It must be one of:

- `S` encoded as:
  U = 0, op = 1, if U is absent
  U = 1, op = 0, if U is present
- `U` encoded as U = 1, op = 1. Not available for `VQSHLU`.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` Encoded as L = '0', imm6<5:3> = '001'. `<imm>` is encoded in imm6<2:0>.
- `16` Encoded as L = '0', imm6<5:4> = '01'. `<imm>` is encoded in imm6<3:0>.
- `32` Encoded as L = '0', imm6<5> = '1'. `<imm>` is encoded in imm6<4:0>.
- `64` Encoded as L = '1'. `<imm>` is encoded in imm6<5:0>.

`<Qd>, <Qm>` The destination vector, and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector, and the operand vector, for a doubleword operation.

`<imm>` The immediate value, in the range 0 to `<size>`-1. See the description of `<size>` for how `<imm>` is encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            operand = Int(Elem[D[m+r],e,esize], src_unsigned);
            (result, sat) = SatQ(operand << shift_amount, esize, dest_unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VQSHL{U}<c><q>.<type><size>  {<Qd>,} <Qm>, #<imm>        Q = 1 としてエンコードされる
VQSHL{U}<c><q>.<type><size>  {<Dd>,} <Dm>, #<imm>        Q = 0 としてエンコードされる
```

各項目の説明については以下を参照して下さい。

`U` 存在する場合、オペランドが符号付きであっても結果を符号なしにすることが指定されます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VQSHL` および `VQSHLU` は常に無条件命令です。

`<type>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S` 次のようにエンコードされます。
  U = 0、op = 1（U を省略した場合）
  U = 1、op = 0（U を指定した場合）
- `U` U = 1、op = 1 としてエンコードされます。`VQSHLU` では使用できません。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` L = '0'、imm6〈5:3〉= '001' としてエンコードされます。`<imm>` は、imm6〈2:0〉にエンコードされます。
- `16` L = '0'、imm6〈5:4〉= '01' としてエンコードされます。`<imm>` は、imm6〈3:0〉にエンコードされます。
- `32` L = '0'、imm6〈5〉= '1' としてエンコードされます。`<imm>` は、imm6〈4:0〉にエンコードされます。
- `64` L = '1' としてエンコードされます。`<imm>` は、imm6〈5:0〉にエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<imm>` 0 ～ `<size>` − 1 の範囲のイミディエート値。`<imm>` のエンコード方法については、`<size>` の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            operand = Int(Elem[D[m+r],e,esize], src_unsigned);
            (result, sat) = SatQ(operand << shift_amount, esize, dest_unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.368 VQSHRN, VQSHRUN

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the truncated results in a doubleword vector.

For rounded results, see *VQRSHRN, VQRSHRUN* on page A8-716.

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQSHR{U}N<c>.<type><size> <Dd>,<Qm>,#<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				1	0	0	op	0	0	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				1	0	0	op	0	0	M	1	Vm			

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE VSHRN;
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.368 VQSHRN、VQSHRUN

VQSHRN、VQSHRUN（ベクタ飽和右シフトおよび縮小）は、整数のクワッドワードベクタの各エレメントを取得して、イミディエート値分だけ右にシフトし、切り捨てられた結果をダブルワードベクタに配置します。

丸められる結果については、*VQRSHRN*、*VQRSHRUN* P. A8-716 を参照して下さい。

オペランドのエレメントはすべて同じサイズで、次のいずれかを使用できます。

- 16 ビット、32 ビット、または 64 ビットの符号付き整数
- 16 ビット、32 ビット、または 64 ビットの符号なし整数

結果のエレメントの幅は、オペランドのエレメントの半分です。オペランドのエレメントが符号付きの場合、結果のエレメントは符号付きまたは符号なしです。オペランドのエレメントが符号なしの場合、結果のエレメントも符号なしです。

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2-9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQSHR{U}N<c>.<type><size> <Dd>,<Qm>,#<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				1	0	0	op	0	0	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				1	0	0	op	0	0	M	1	Vm			

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE VSHRN;
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VQSHR{U}N<c><q>.<type><size>  <Dd>, <Qm>, #<imm>
```

where:

U — If present, specifies that the results are unsigned, although the operands are signed.

<c><q> — See *Standard assembler syntax fields* on page A8-7. An ARM VQSHRN or VQSHRUN instruction must be unconditional.

<type> — The data type for the elements of the vectors. It must be one of:

- S — encoded as:
  - U = 0, op = 1, if U is absent
  - U = 1, op = 0, if U is present
- U — encoded as U = 1, op = 1. Not available for VQSHRUN.

<size> — The data size for the elements of the vectors. It must be one of:

- 16 — Encoded as imm6<5:3> = '001'. (8 – <imm>) is encoded in imm6<2:0>.
- 32 — Encoded as imm6<5:4> = '01'. (16 – <imm>) is encoded in imm6<3:0>.
- 64 — Encoded as imm6<5> = '1'. (32 – <imm>) is encoded in imm6<4:0>.

<Dd>, <Qm> — The destination vector, and the operand vector.

<imm> — The immediate value, in the range 1 to <size>/2. See the description of <size> for how <imm> is encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        operand = Int(Elem[Q[m>>1],e,2*esize], src_unsigned);
        (result, sat) = SatQ(operand >> shift_amount, esize, dest_unsigned);
        Elem[D[d],e,esize] = result;
        if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

Pseudo-instructions

VQSHRN.I<size> <Dd>, <Qm>, #0	is a synonym for	VQMOVN.I<size> <Dd>, <Qm>
VQSHRUN.I<size> <Dd>, <Qm>, #0	is a synonym for	VQMOVUN.I<size> <Dd>, <Qm>

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VQSHR{U}N<c><q>.<type><size>  <Dd>, <Qm>, #<imm>
```

各項目の説明については以下を参照して下さい。

U — 存在する場合、オペランドが符号付きであっても結果を符号なしにすることが指定されます。

<c><q> — *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM VQSHRN および VQSHRUN は常に無条件命令です。

<type> — ベクタのエレメントのデータ型。次のいずれかを使用できます。

- S — 次のようにエンコードされます。
  - U = 0、op = 1（U を省略した場合）
  - U = 1、op = 0（U を指定した場合）
- U — U = 1、op = 1 としてエンコードされます。VQSHRUN では使用できません。

<size> — ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- 16 — imm6〈5:3〉= '001' としてエンコードされます。(8 - <imm>) は、imm6〈2:0〉にエンコードされます。
- 32 — imm6〈5:4〉= '01' としてエンコードされます。(16 - <imm>) は、imm6〈3:0〉にエンコードされます。
- 64 — imm6〈5〉= '1' としてエンコードされます。(32 - <imm>) は、imm6〈4:0〉にエンコードされます。

<Dd>, <Qm> — デスティネーションベクタとオペランドベクタ

<imm> — 1 ～ <size> / 2 の範囲のイミディエート値。<imm> のエンコード方法については、<size> の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        operand = Int(Elem[Q[m>>1],e,2*esize], src_unsigned);
        (result, sat) = SatQ(operand >> shift_amount, esize, dest_unsigned);
        Elem[D[d],e,esize] = result;
        if sat then FPSCR.QC = '1';
```

例外

未定義命令

擬似命令

VQSHRN.I<size> <Dd>, <Qm>, #0	は次の命令と同義語	VQMOVN.I<size> <Dd>, <Qm>
VQSHRUN.I<size> <Dd>, <Qm>, #0	は次の命令と同義語	VQMOVUN.I<size> <Dd>, <Qm>

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.369 VQSUB

Vector Saturating Subtract subtracts the elements of the second operand vector from the corresponding elements of the first operand vector, and places the results in the destination vector. Signed and unsigned operations are distinct.

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

If any of the results overflow, they are saturated. The cumulative saturation flag, QC, is set if saturation occurs. For details see *Pseudocode details of saturation* on page A2-9.

Encoding T1 / A1 Advanced SIMD

```
VQSUB<c>.<type><size> <Qd>, <Qn>, <Qm>
VQSUB<c>.<type><size> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	1	0	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	1	0	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.369 VQSUB

VQSUB（ベクタ飽和減算）は、2 番目のオペランドベクタのエレメントを、最初のオペランドベクタの対応するエレメントから減算し、結果をデスティネーションベクタに配置します。演算には符号付きと符号なしの区別があります。

オペランドと結果のエレメントのデータ型はすべて同じで、次のいずれかを使用できます。

- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号付き整数
- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号なし整数

結果がオーバフローした場合は飽和されます。飽和が発生すると、累積飽和フラグ QC がセットされます。詳細については、*飽和の擬似コードの詳細* P. A2- 9 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VQSUB<c>.<type><size> <Qd>, <Qn>, <Qm>
VQSUB<c>.<type><size> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	1	0	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	1	0	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VQSUB<c><q>.<type><size> {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VQSUB<c><q>.<type><size> {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VQSUB` instruction must be unconditional.

`<type>` The data type for the elements of the vectors. It must be one of:

- `S` signed, encoded as U = 0
- `U` unsigned, encoded as U = 1.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10
- `64` encoded as size = 0b11.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            diff = Int(Elem[D[n+r],e,esize], unsigned) - Int(Elem[D[m+r],e,esize], unsigned);
            (Elem[D[d+r],e,esize], sat) = SatQ(diff, esize, unsigned);
            if sat then FPSCR.QC = '1';
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VQSUB<c><q>.<type><size> {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VQSUB<c><q>.<type><size> {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VQSUB` は常に無条件命令です。

`<type>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S` 符号付き、U = 0 としてエンコードされます。
- `U` 符号なし、U = 1 としてエンコードされます。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。
- `64` size = 0b11 としてエンコードされます。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            diff = Int(Elem[D[n+r],e,esize], unsigned) - Int(Elem[D[m+r],e,esize], unsigned);
            (Elem[D[d+r],e,esize], sat) = SatQ(diff, esize, unsigned);
            if sat then FPSCR.QC = '1';
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.370 VRADDHN

Vector Rounding Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are rounded. (For truncated results, see *VADDHN* on page A8-540.)

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Encoding T1 / A1 Advanced SIMD

```
VRADDHN<c>.<dt> <Dd>, <Qn>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	size		Vn				Vd				0	1	0	0	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	size		Vn				Vd				0	1	0	0	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.370 VRADDHN

VRADDHN（上位半分を返すベクタ丸め付き加算および縮小）は、2 つのクワッドワードベクタの対応するエレメントを加算し、各結果の上位半分をダブルワードベクタに配置します。結果は丸められます（切り捨てられる結果については、*VADDHN* P. A8- 540 を参照して下さい）。

オペランドのエレメントには、16 ビット、32 ビット、または 64 ビットの整数を使用できます。整数には符号付きと符号なしの区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VRADDHN<c>.<dt> <Dd>, <Qn>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	size		Vn				Vd				0	1	0	0	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	size		Vn				Vd				0	1	0	0	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7- 10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VRADDHN<c><q>.<dt>  <Dd>, <Qn>, <Qm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM `VRADDHN` instruction must be unconditional.
`<dt>`	The data type for the elements of the operands. It must be one of: `I16` size = 0b00 `I32` size = 0b01 `I64` size = 0b10.
`<Dd>, <Qn>, <Qm>`	The destination vector and the operand vectors.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Q[n>>1],e,2*esize] + Elem[Q[m>>1],e,2*esize] + round_const;
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VRADDHN<c><q>.<dt>  <Dd>, <Qn>, <Qm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VRADDHN` は常に無条件命令です。
`<dt>`	オペランドのエレメントのデータ型。次のいずれかを使用できます。 `I16` size = 0b00 `I32` size = 0b01 `I64` size = 0b10
`<Dd>, <Qn>, <Qm>`	デスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Q[n>>1],e,2*esize] + Elem[Q[m>>1],e,2*esize] + round_const;
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.371 VRECPE

Vector Reciprocal Estimate finds an approximate reciprocal of each element in the operand vector, and places the results in the destination vector.

The operand and result elements are the same type, and can be 32-bit floating-point numbers, or 32-bit unsigned integers.

For details of the operation performed by this instruction see *Reciprocal estimate and step* on page A2-58.

Encoding T1 / A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

```
VRECPE<c>.<dt> <Qd>, <Qm>
VRECPE<c>.<dt> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	1	Vd				0	1	0	F	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	1	Vd				0	1	0	F	0	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
floating_point = (F == '1');  esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.371 VRECPE

VRECPE（ベクタ逆数概算）は、オペランドベクタの各エレメントの近似逆数を求め、結果をデスティネーションベクタに配置します。

オペランドと結果のエレメントは同じデータ型で、32 ビットの浮動小数点数または 32 ビットの符号なし整数を使用できます。

この命令によって実行される演算の詳細については、*逆数概算とステップ* P. A2- 58 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは F = 1 は未定義）

```
VRECPE<c>.<dt> <Qd>, <Qm>
VRECPE<c>.<dt> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	1	Vd				0	1	0	F	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	1	Vd				0	1	0	F	0	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
floating_point = (F == '1');  esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VRECPE<c><q>.<dt>  <Qd>, <Qm>`	Encoded as Q = 1
`VRECPE<c><q>.<dt>  <Dd>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VRECPE` instruction must be unconditional.

`<dt>` The data types for the elements of the vectors. It must be one of:

- `U32` encoded as F = 0, size = 0b10
- `F32` encoded as F = 1, size = 0b10.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                Elem[D[d+r],e,esize] = FPRecipEstimate(Elem[D[m+r],e,esize]);
            else
                Elem[D[d+r],e,esize] = UnsignedRecipEstimate(Elem[D[m+r],e,esize]);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, Underflow, and Division by Zero.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see *Reciprocal estimate and step* on page A2-58.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VRECPE<c><q>.<dt>  <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VRECPE<c><q>.<dt>  <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VRECPE` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `U32` F = 0、size = 0b10 としてエンコードされます。
- `F32` F = 1、size = 0b10 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                Elem[D[d+r],e,esize] = FPRecipEstimate(Elem[D[m+r],e,esize]);
            else
                Elem[D[d+r],e,esize] = UnsignedRecipEstimate(Elem[D[m+r],e,esize]);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、アンダーフロー、0 による除算

ニュートン・ラプソン反復法

実行される演算と、ニュートン・ラプソン反復法でその演算を使用して数値の逆数を求める方法の詳細については、*逆数概算とステップ* P. A2-58 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.372 VRECPS

Vector Reciprocal Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 2.0, and places the results into the elements of the destination vector.

The operand and result elements are 32-bit floating-point numbers.

For details of the operation performed by this instruction see *Reciprocal estimate and step* on page A2-58.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
VRECPS<c>.F32 <Qd>, <Qn>, <Qm>
VRECPS<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.372 VRECPS

VRECPS（ベクタ逆数ステップ）は、一方のベクタのエレメントと他方のベクタの対応するエレメントを乗算し、それぞれの積を 2.0 から減算し、結果をデスティネーションベクタのエレメントに配置します。

オペランドと結果のエレメントは、32 ビットの浮動小数点数です。

この命令によって実行される演算の詳細については、*逆数概算とステップ* P. A2- 58 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VRECPS<c>.F32 <Qd>, <Qn>, <Qm>
VRECPS<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VRECPS<c><q>.F32 {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VRECPS<c><q>.F32 {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM `VRECPS` instruction must be unconditional.
`<Qd>, <Qn>, <Qm>`	The destination vector and the operand vectors for a quadword operation.
`<Dd>, <Dn>, <Dm>`	The destination vector and the operand vectors for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPRecipStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, Overflow, Underflow, and Inexact.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see *Reciprocal estimate and step* on page A2-58.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VRECPS<c><q>.F32 {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VRECPS<c><q>.F32 {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VRECPS` は常に無条件命令です。
`<Qd>, <Qn>, <Qm>`	クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Dd>, <Dn>, <Dm>`	ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPRecipStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、オーバフロー、アンダーフロー、不正確

ニュートン・ラプソン反復法

実行される演算と、ニュートン・ラプソン反復法でその演算を使用して数値の逆数を求める方法の詳細については、*逆数概算とステップ* P. A2-58 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.373 VREV16, VREV32, VREV64

VREV16 (Vector Reverse in halfwords) reverses the order of 8-bit elements in each halfword of the vector, and places the result in the corresponding destination vector.

VREV32 (Vector Reverse in words) reverses the order of 8-bit or 16-bit elements in each word of the vector, and places the result in the corresponding destination vector.

VREV64 (Vector Reverse in doublewords) reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

Encoding T1 / A1 Advanced SIMD

```
VREV<n><c>.<size> <Qd>, <Qm>
VREV<n><c>.<size> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	0	0	op		Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	0	0	op		Q	M	0	Vm			

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
groupsize = (1 << (3-UInt(op)-UInt(size)));  // elements per reversing group: 2, 4 or 8
reverse_mask = (groupsize-1)<esize-1:0>;    // EORing mask used for index calculations
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Figure A8-6 shows two examples of the operation of VREV.

Figure A8-6 Examples of operation

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.373 VREV16、VREV32、VREV64

VREV16（ベクタハーフワード反転）は、ベクタの各ハーフワードにおける 8 ビットのエレメントの順序を反転し、結果を対応するデスティネーションベクタに配置します。

VREV32（ベクタワード反転）は、ベクタの各ワードにおける 8 ビットまたは 16 ビットのエレメントの順序を反転し、結果を対応するデスティネーションベクタに配置します。

VREV64（ベクタダブルワード反転）は、ベクタの各ダブルワードにおける 8 ビット、16 ビット、または 32 ビットのエレメントの順序を反転し、結果を対応するデスティネーションベクタに配置します。

エレメントには、サイズを除きデータ型の区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VREV<n><c>.<size> <Qd>, <Qm>
VREV<n><c>.<size> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	0	0	op		Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	0	0	op		Q	M	0	Vm			

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
groupsize = (1 << (3-UInt(op)-UInt(size)));  // elements per reversing group: 2, 4 or 8
reverse_mask = (groupsize-1)<esize-1:0>;    // EORing mask used for index calculations
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

VREV の動作の例を 2 つ、図 A8-6 に示します。

図 A8-6 動作の例

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VREV<n><c><q>.<size>  <Qd>, <Qm>`	Encoded as Q = 1
`VREV<n><c><q>.<size>  <Dd>, <Dm>`	Encoded as Q = 0

where:

`<n>` The size of the regions in which the vector elements are reversed. It must be one of:

- `16` encoded as op = 0b10
- `32` encoded as op = 0b01
- `64` encoded as op = 0b00.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VREV` instruction must be unconditional.

`<size>` The size of the vector elements. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

`<size>` must specify a smaller size than `<n>`.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

If op + size >= 3, the instruction is reserved.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;

    for r = 0 to regs-1
        for e = 0 to elements-1
            // Calculate destination element index by bitwise EOR on source element index:
            e_bits = e<esize-1:0>;  d_bits = e_bits EOR reverse_mask;  d = UInt(d_bits);
            Elem[dest,d,esize] = Elem[D[m+r],e,esize];
        D[d+r] = dest;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VREV<n><c><q>.<size>  <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VREV<n><c><q>.<size>  <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<n>` ベクタのエレメントが反転される領域のサイズ。次のいずれかを使用できます。

- `16` op = 0b10 としてエンコードされます。
- `32` op = 0b01 としてエンコードされます。
- `64` op = 0b00 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VREV` は常に無条件命令です。

`<size>` ベクタのエレメントのサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

`<size>` には、`<n>` より小さいサイズを指定する必要があります。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

op + size >= 3 の場合、命令は予約されています。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;

    for r = 0 to regs-1
        for e = 0 to elements-1
            // Calculate destination element index by bitwise EOR on source element index:
            e_bits = e<esize-1:0>;  d_bits = e_bits EOR reverse_mask;  d = UInt(d_bits);
            Elem[dest,d,esize] = Elem[D[m+r],e,esize];
        D[d+r] = dest;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.374 VRHADD

Vector Rounding Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector.

The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers
- 8-bit, 16-bit, or 32-bit unsigned integers.

The results of the halving operations are rounded (for truncated results see *VHADD, VHSUB* on page A8-600).

Encoding T1 / A1 Advanced SIMD

```
VRHADD<c> <Qd>, <Qn>, <Qm>
VRHADD<c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	0	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.374 VRHADD

VRHADD（ベクタ丸め付き加算半減）は、整数の 2 つのベクタの対応するエレメントを加算し、各結果を右に 1 ビットシフトし、最終結果をデスティネーションベクタに配置します。

オペランドと結果のエレメントのデータ型はすべて同じで、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットの符号付き整数
- 8 ビット、16 ビット、または 32 ビットの符号なし整数

半減演算の結果は丸められます（切り捨てられる結果については、*VHADD*、*VHSUB* P. A8- 600 を参照して下さい）。

エンコード T1/A1 アドバンスト SIMD

```
VRHADD<c> <Qd>, <Qn>, <Qm>
VRHADD<c> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	0	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VRHADD<c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VRHADD<c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VRHADD` instruction must be unconditional.

`<dt>` The data type for the elements of the vectors. It must be one of:

- `S8` encoded as size = 0b00, U = 0
- `S16` encoded as size = 0b01, U = 0
- `S32` encoded as size = 0b10, U = 0
- `U8` encoded as size = 0b00, U = 1
- `U16` encoded as size = 0b01, U = 1
- `U32` encoded as size = 0b10, U = 1.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = op1 + op2 + 1;
            Elem[D[d+r],e,esize] = result<esize:1>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VRHADD<c><q>.<dt> {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VRHADD<c><q>.<dt> {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VRHADD` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S8` size = 0b00、U = 0 としてエンコードされます。
- `S16` size = 0b01、U = 0 としてエンコードされます。
- `S32` size = 0b10、U = 0 としてエンコードされます。
- `U8` size = 0b00、U = 1 としてエンコードされます。
- `U16` size = 0b01、U = 1 としてエンコードされます。
- `U32` size = 0b10、U = 1 としてエンコードされます。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = op1 + op2 + 1;
            Elem[D[d+r],e,esize] = result<esize:1>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.375 VRSHL

Vector Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. (For a truncating shift, see *VSHL (register)* on page A8-752).

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Encoding T1 / A1 Advanced SIMD

```
VRSHL<c>.<type><size> <Qd>, <Qm>, <Qn>
VRSHL<c>.<type><size> <Dd>, <Dm>, <Dn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	0	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.375 VRSHL

VRSHL（ベクタ丸め付き左シフト）は、ベクタの各エレメントを取得して、2 番目のベクタの対応するエレメントの最下位バイトから取得された値で示されるビット数だけシフトし、結果をデスティネーションベクタに配置します。シフトの値が正の場合、演算は左シフトになります。シフトの値が負の場合、演算は丸め付き右シフトになります（切り捨て付きシフトについては、*VSHL（レジスタ）* P. A8- 752 を参照して下さい）。

最初のオペランドと結果のエレメントのデータ型は同じで、次のいずれかを使用できます。

- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号付き整数
- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号なし整数

2 番目のオペランドは、同じサイズの符号付き整数です。

エンコード T1/A1 アドバンスト SIMD

```
VRSHL<c>.<type><size> <Qd>, <Qm>, <Qn>
VRSHL<c>.<type><size> <Dd>, <Dm>, <Dn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	0	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VRSHL<c><q>.<type><size> {<Qd>,} <Qm>, <Qn>`	Encoded as Q = 1
`VRSHL<c><q>.<type><size> {<Dd>,} <Dm>, <Dn>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VRSHL` instruction must be unconditional.

`<type>` The data type for the elements of the vectors. It must be one of:

- `S` signed, encoded as U = 0
- `U` unsigned, encoded as U = 1.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10
- `64` encoded as size = 0b11.

`<Qd>, <Qm>, <Qn>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dm>, <Dn>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            round_const = 1 << (-shift-1); // 0 for left shift, 2^(n-1) for right shift
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) << shift;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VRSHL<c><q>.<type><size> {<Qd>,} <Qm>, <Qn>`	Q = 1 としてエンコードされる
`VRSHL<c><q>.<type><size> {<Dd>,} <Dm>, <Dn>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VRSHL` は常に無条件命令です。

`<type>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S` 符号付き、U = 0 としてエンコードされます。
- `U` 符号なし、U = 1 としてエンコードされます。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。
- `64` size = 0b11 としてエンコードされます。

`<Qd>, <Qm>, <Qn>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>, <Dn>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            round_const = 1 << (-shift-1); // 0 for left shift, 2^(n-1) for right shift
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) << shift;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.376 VRSHR

Vector Rounding Shift Right takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see *VSHR* on page A8-756.

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Encoding T1 / A1 Advanced SIMD

```
VRSHR<c>.<type><size> <Qd>, <Qm>, #<imm>
VRSHR<c>.<type><size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	1	0	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	1	0	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.376 VRSHR

VRSHR（ベクタ丸め付き右シフト）は、ベクタの各エレメントを取得して、イミディエート値だけ右にシフトし、丸められた結果をデスティネーションベクタに配置します。切り捨てられる結果については、*VSHR* P. A8-756 を参照して下さい。

オペランドと結果のエレメントのサイズは同じで、次のいずれかを使用できます。

- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号付き整数
- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号なし整数

エンコード T1/A1 アドバンスト SIMD

```
VRSHR<c>.<type><size> <Qd>, <Qm>, #<imm>
VRSHR<c>.<type><size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	1	0	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	1	0	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VRSHR<c><q>.<type><size>  {<Qd>,} <Qm>, #<imm>        Encoded as Q = 1
VRSHR<c><q>.<type><size>  {<Dd>,} <Dm>, #<imm>        Encoded as Q = 0
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VRSHR` instruction must be unconditional.

`<type>` The data type for the elements of the vectors. It must be one of:

- `S` signed, encoded as U = 0
- `U` unsigned, encoded as U = 1.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` Encoded as L = '0', imm6<5:3> = '001'. (8– `<imm>`) is encoded in imm6<2:0>.
- `16` Encoded as L = '0', imm6<5:4> = '01'. (16 – `<imm>`) is encoded in imm6<3:0>.
- `32` Encoded as L = '0', imm6<5> = '1'. (32 – `<imm>`) is encoded in imm6<4:0>.
- `64` Encoded as L = '1'. (64 – `<imm>`) is encoded in imm6<5:0>.

`<Qd>, <Qm>` The destination vector, and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector, and the operand vector, for a doubleword operation.

`<imm>` The immediate value, in the range 1 to `<size>`. See the description of `<size>` for how `<imm>` is encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> shift_amount;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Exceptions

Undefined Instruction.

Pseudo-instructions

```
VRSHR.<type><size>  <Qd>, <Qm>, #0     is a synonym for     VMOV <Qd>, <Qm>
VRSHR.<type><size>  <Dd>, <Dm>, #0     is a synonym for     VMOV <Dd>, <Dm>
```

For details see *VMOV (register)* on page A8-642.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VRSHR<c><q>.<type><size>  {<Qd>,} <Qm>, #<imm>        Q = 1 としてエンコードされる
VRSHR<c><q>.<type><size>  {<Dd>,} <Dm>, #<imm>        Q = 0 としてエンコードされる
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VRSHR` は常に無条件命令です。

`<type>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S` 符号付き、U = 0 としてエンコードされます。
- `U` 符号なし、U = 1 としてエンコードされます。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` L = '0'、imm6〈5:3〉 = '001' としてエンコードされます。(8 - `<imm>`) は、imm6〈2:0〉 にエンコードされます。
- `16` L = '0'、imm6〈5:4〉 = '01' としてエンコードされます。(16 - `<imm>`) は、imm6〈3:0〉 にエンコードされます。
- `32` L = '0'、imm6〈5〉 = '1' としてエンコードされます。(32 - `<imm>`) は、imm6〈4:0〉 にエンコードされます。
- `64` L = '1' としてエンコードされます。(64 - `<imm>`) は、imm6〈5:0〉 にエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<imm>` 1 ～ `<size>` の範囲のイミディエート値。`<imm>` のエンコード方法については、`<size>` の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> shift_amount;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

例外

未定義命令

擬似命令

```
VRSHR.<type><size>  <Qd>, <Qm>, #0     は次の命令と同義語     VMOV <Qd>, <Qm>
VRSHR.<type><size>  <Dd>, <Dm>, #0     は次の命令と同義語     VMOV <Dd>, <Dm>
```

詳細については、*VMOV (レジスタ)* P. A8-642 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.377 VRSHRN

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see *VSHRN* on page A8-758.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Encoding T1 / A1 Advanced SIMD

```
VRSHRN<c>.I<size> <Dd>, <Qm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	imm6						Vd				1	0	0	0	0	1	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	imm6						Vd				1	0	0	0	0	1	M	1	Vm			

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx' then esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx' then esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx' then esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.377 VRSHRN

VRSHRN（ベクタ丸め付き右シフトおよび縮小）は、ベクタの各エレメントを取得して、イミディエート値分だけ右にシフトし、丸められた結果をデスティネーションベクタに配置します。切り捨てられる結果については、*VSHRN* P. A8-758 を参照して下さい。

オペランドのエレメントには、16 ビット、32 ビット、または 64 ビットの整数を使用できます。整数には符号付きと符号なしの区別はありません。デスティネーションのエレメントのサイズは、ソースのエレメントの半分です。

エンコード T1/A1 アドバンスト SIMD

```
VRSHRN<c>.I<size> <Dd>, <Qm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	imm6						Vd				1	0	0	0	0	1	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	imm6						Vd				1	0	0	0	0	1	M	1	Vm			

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx' then esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx' then esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx' then esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VRSHRN<c><q>.I<size>  <Dd>, <Qm>, #<imm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7. An ARM VRSHRN instruction must be unconditional.

<size> The data size for the elements of the vectors. It must be one of:

- 16 Encoded as imm6<5:3> = '001'. (8 – <imm>) is encoded in imm6<2:0>.
- 32 Encoded as imm6<5:4> = '01'. (16 – <imm>) is encoded in imm6<3:0>.
- 64 Encoded as imm6<5> = '1'. (32 – <imm>) is encoded in imm6<4:0>.

<Dd>, <Qm> The destination vector, and the operand vector.

<imm> The immediate value, in the range 1 to <size>/2. See the description of <size> for how <imm> is encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount-1);
    for e = 0 to elements-1
        result = LSR(Elem[Q[m>>1],e,2*esize] + round_const, shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```

Exceptions

Undefined Instruction.

Pseudo-instructions

VRSHRN.I<size> <Dd>, <Qm>, #0 is a synonym for VMOVN.I<size> <Dd>, <Qm>

For details see *VMOVN* on page A8-656.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VRSHRN<c><q>.I<size>  <Dd>, <Qm>, #<imm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM VRSHRN は常に無条件命令です。

<size> ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- 16 imm6〈5:3〉= '001' としてエンコードされます。(8 - <imm>) は、imm6〈2:0〉にエンコードされます。
- 32 imm6〈5:4〉= '01' としてエンコードされます。(16 - <imm>) は、imm6〈3:0〉にエンコードされます。
- 64 imm6〈5〉= '1' としてエンコードされます。(32 - <imm>) は、imm6〈4:0〉にエンコードされます。

<Dd>, <Qm> デスティネーションベクタとオペランドベクタ

<imm> 1 ～ <size> / 2 の範囲のイミディエート値。<imm> のエンコード方法については、<size> の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount-1);
    for e = 0 to elements-1
        result = LSR(Elem[Q[m>>1],e,2*esize] + round_const, shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```

例外

未定義命令

擬似命令

VRSHRN.I<size> <Dd>, <Qm>, #0 は次の命令と同義語VMOVN.I<size> <Dd>, <Qm>

詳細については、*VMOVN* P. A8-656 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.378 VRSQRTE

Vector Reciprocal Square Root Estimate finds an approximate reciprocal square root of each element in a vector, and places the results in a second vector.

The operand and result elements are the same type, and can be 32-bit floating-point numbers, or 32-bit unsigned integers.

For details of the operation performed by this instruction see *Reciprocal square root* on page A2-61.

Encoding T1 / A1 Advanced SIMD (F = 1 UNDEFINED in integer-only variants)

```
VRSQRTE<c>.<dt> <Qd>, <Qm>
VRSQRTE<c>.<dt> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	1	Vd				0	1	0	F	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	1	Vd				0	1	0	F	1	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
floating_point = (F == '1');  esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.378 VRSQRTE

VRSQRTE（ベクタ逆数平方根概算）は、ベクタの各エレメントの近似逆数平方根を求め、結果を 2 番目のベクタに配置します。

オペランドと結果のエレメントは同じデータ型で、32 ビットの浮動小数点数または 32 ビットの符号なし整数を使用できます。

この命令によって実行される演算の詳細については、*逆数平方根* P. A2-61 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは F = 1 は未定義）

```
VRSQRTE<c>.<dt> <Qd>, <Qm>
VRSQRTE<c>.<dt> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	1	Vd				0	1	0	F	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	1	Vd				0	1	0	F	1	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size != '10' then UNDEFINED;
floating_point = (F == '1');  esize = 32;  elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VRSQRTE<c><q>.<dt>  <Qd>, <Qm>`	Encoded as Q = 1
`VRSQRTE<c><q>.<dt>  <Dd>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VRSQRTE` instruction must be unconditional.

`<dt>` The data types for the elements of the vectors. It must be one of:

- `U32` encoded as F = 0, size = 0b10
- `F32` encoded as F = 1, size = 0b10.

`<Qd>, <Qm>` The destination vector and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                Elem[D[d+r],e,esize] = FPRSqrtEstimate(Elem[D[m+r],e,esize]);
            else
                Elem[D[d+r],e,esize] = UnsignedRSqrtEstimate(Elem[D[m+r],e,esize]);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, and Division by Zero.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see *Reciprocal square root* on page A2-61.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VRSQRTE<c><q>.<dt>  <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VRSQRTE<c><q>.<dt>  <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VRSQRTE` は常に無条件命令です。

`<dt>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `U32` F = 0、size = 0b10 としてエンコードされます。
- `F32` F = 1、size = 0b10 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                Elem[D[d+r],e,esize] = FPRSqrtEstimate(Elem[D[m+r],e,esize]);
            else
                Elem[D[d+r],e,esize] = UnsignedRSqrtEstimate(Elem[D[m+r],e,esize]);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、0 による除算

ニュートン・ラプソン反復法

実行される演算と、ニュートン・ラプソン反復法でその演算を使用して数値の平方根の逆数を求める方法の詳細については、*逆数平方根* P. A2-61 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.379 VRSQRTS

Vector Reciprocal Square Root Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 3.0, divides these results by 2.0, and places the results into the elements of the destination vector.

The operand and result elements are 32-bit floating-point numbers.

For details of the operation performed by this instruction see *Reciprocal square root* on page A2-61.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
VRSQRTS<c>.F32 <Qd>, <Qn>, <Qm>
VRSQRTS<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.379 VRSQRTS

VRSQRTS（ベクタ逆数平方根ステップ）は、一方のベクタのエレメントと他方のベクタの対応するエレメントを乗算し、それぞれの積を 3.0 から減算し、それらの結果を 2.0 で除算し、結果をデスティネーションベクタのエレメントに配置します。

オペランドと結果のエレメントは、32 ビットの浮動小数点数です。

この命令によって実行される演算の詳細については、*逆数平方根* P. A2- 61 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VRSQRTS<c>.F32 <Qd>, <Qn>, <Qm>
VRSQRTS<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VRSQRTS<c><q>.F32 {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1, sz = 0
`VRSQRTS<c><q>.F32 {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0, sz = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VRSQRTS` instruction must be unconditional.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPRSqrtStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, Overflow, Underflow, and Inexact.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see *Reciprocal square root* on page A2-61.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VRSQRTS<c><q>.F32 {<Qd>,} <Qn>, <Qm>`	Q = 1、sz = 0 としてエンコードされる
`VRSQRTS<c><q>.F32 {<Dd>,} <Dn>, <Dm>`	Q = 0、sz = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VRSQRTS` は常に無条件命令です。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPRSqrtStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、オーバフロー、アンダーフロー、不正確

ニュートン・ラプソン反復法

実行される演算と、ニュートン・ラプソン反復法でその演算を使用して数値の平方根の逆数を求める方法の詳細については、*逆数平方根* P. A2-61 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.380 VRSRA

Vector Rounding Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the rounded results into the destination vector. (For truncated results, see *VSRA* on page A8-764.)

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Encoding T1 / A1 Advanced SIMD

```
VRSRA<c>.<type><size> <Qd>, <Qm>, #<imm>
VRSRA<c>.<type><size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	1	1	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	1	1	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.380 VRSRA

VRSRA（ベクタ丸め付き右シフトおよび累積）は、ベクタの各エレメントを取得して、イミディエート値分だけ右にシフトし、丸められた結果をデスティネーションベクタに累積します（切り捨てられる結果については、*VSRA* P. A8-764 を参照して下さい）。

オペランドと結果のエレメントのデータ型はすべて同じで、次のいずれかを使用できます。

- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号付き整数
- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号なし整数

エンコード T1/A1 アドバンスト SIMD

```
VRSRA<c>.<type><size> <Qd>, <Qm>, #<imm>
VRSRA<c>.<type><size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	1	1	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	1	1	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VRSRA<c><q>.<type><size> {<Qd>,} <Qm>, #<imm>`	Encoded as Q = 1
`VRSRA<c><q>.<type><size> {<Dd>,} <Dm>, #<imm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VRSRA` instruction must be unconditional.

`<type>` The data type for the elements of the vectors. It must be one of:

- `S` signed, encoded as U = 0
- `U` unsigned, encoded as U = 1.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` Encoded as L = '0', imm6<5:3> = '001'. (8– `<imm>`) is encoded in imm6<2:0>.
- `16` Encoded as L = '0', imm6<5:4> = '01'. (16 – `<imm>`) is encoded in imm6<3:0>.
- `32` Encoded as L = '0', imm6<5> = '1'. (32 – `<imm>`) is encoded in imm6<4:0>.
- `64` Encoded as L = '1'. (64 – `<imm>`) is encoded in imm6<5:0>.

`<Qd>, <Qm>` The destination vector, and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector, and the operand vector, for a doubleword operation.

`<imm>` The immediate value, in the range 1 to `<size>`. See the description of `<size>` for how `<imm>` is encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> shift_amount;
            Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VRSRA<c><q>.<type><size> {<Qd>,} <Qm>, #<imm>`	Q = 1 としてエンコードされる
`VRSRA<c><q>.<type><size> {<Dd>,} <Dm>, #<imm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。ARM `VRSRA` は常に無条件命令です。

`<type>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S` 符号付き、U = 0 としてエンコードされます。
- `U` 符号なし、U = 1 としてエンコードされます。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` L = '0'、imm6〈5:3〉= '001' としてエンコードされます。(8 - `<imm>`) は、imm6〈2:0〉にエンコードされます。
- `16` L = '0'、imm6〈5:4〉= '01' としてエンコードされます。(16 - `<imm>`) は、imm6〈3:0〉にエンコードされます。
- `32` L = '0'、imm6〈5〉= '1' としてエンコードされます。(32 - `<imm>`) は、imm6〈4:0〉にエンコードされます。
- `64` L = '1' としてエンコードされます。(64 - `<imm>`) は、imm6〈5:0〉にエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<imm>` 1 ～ `<size>` の範囲のイミディエート値。`<imm>` のエンコード方法については、`<size>` の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> shift_amount;
            Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.381 VRSUBHN

Vector Rounding Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector takes the most significant half of each result, and places the final results in a doubleword vector. The results are rounded. (For truncated results, see *VSUBHN* on page A8-792.)

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Encoding T1 / A1 Advanced SIMD

```
VRSUBHN<c>.<dt> <Dd>, <Qn>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	size		Vn				Vd				0	1	1	0	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	size		Vn				Vd				0	1	1	0	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.381 VRSUBHN

VRSUBHN（上位半分を返すベクタ丸め付き減算および縮小）は、一方のクワッドワードベクタのエレメントを、他方のクワッドワードベクタの対応するエレメントから減算し、各結果の上位半分を取得し、最終結果をダブルワードベクタに配置します。結果は丸められます（切り捨てられる結果については、*VSUBHN* P. A8-792 を参照して下さい）。

オペランドのエレメントには、16 ビット、32 ビット、または 64 ビットの整数を使用できます。整数には符号付きと符号なしの区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VRSUBHN<c>.<dt> <Dd>, <Qn>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	size		Vn				Vd				0	1	1	0	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	size		Vn				Vd				0	1	1	0	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VRSUBHN<c><q>.<dt>  <Dd>, <Qn>, <Qm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VRSUBHN` instruction must be unconditional.

`<dt>` The data type for the elements of the operands. It must be one of:

`I16`	size = 0b00
`I32`	size = 0b01
`I64`	size = 0b10.

`<Dd>, <Qn>, <Qm>` The destination vector and the operand vectors.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Q[n>>1],e,2*esize] - Elem[Q[m>>1],e,2*esize] + round_const;
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VRSUBHN<c><q>.<dt>  <Dd>, <Qn>, <Qm>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VRSUBHN` は常に無条件命令です。

`<dt>` オペランドのエレメントのデータ型。次のいずれかを使用できます。

`I16`	size = 0b00
`I32`	size = 0b01
`I64`	size = 0b10

`<Dd>, <Qn>, <Qm>` デスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Q[n>>1],e,2*esize] - Elem[Q[m>>1],e,2*esize] + round_const;
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.382 VSHL (immediate)

Vector Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector.

Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Encoding T1 / A1 Advanced SIMD

```
VSHL<c>.I<size> <Qd>, <Qm>, #<imm>
VSHL<c>.I<size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings See *One register and a modified immediate value* on page A7-21

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.382 VSHL（イミディエート）

VSHL（ベクタ左シフト（イミディエート））は、整数のベクタの各エレメントを取得して、イミディエート値だけ左にシフトし、結果をデスティネーションベクタに配置します。

各エレメントの左のシフトアウトされたビットは失われます。

エレメントはすべて同じサイズで、8 ビット、16 ビット、32 ビット、または 64 ビットの整数を使用できます。整数には符号付きと符号なしの区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VSHL<c>.I<size> <Qd>, <Qm>, #<imm>
VSHL<c>.I<size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VSHL<c><q>.I<size> {<Qd>,} <Qm>, #<imm>`	Encoded as Q = 1
`VSHL<c><q>.I<size> {<Dd>,} <Dm>, #<imm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VSHL` instruction must be unconditional.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` Encoded as L = '0', imm6<5:3> = '001'. `<imm>` is encoded in imm6<2:0>.
- `16` Encoded as L = '0', imm6<5:4> = '01'. `<imm>` is encoded in imm6<3:0>.
- `32` Encoded as L = '0', imm6<5> = '1'. `<imm>` is encoded in imm6<4:0>.
- `64` Encoded as L = '1'. `<imm>` is encoded in imm6<5:0>.

`<Qd>, <Qm>` The destination vector, and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector, and the operand vector, for a doubleword operation.

`<imm>` The immediate value, in the range 0 to `<size>`-1. See the description of `<size>` for how `<imm>` is encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = LSL(Elem[D[m+r],e,esize], shift_amount);
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VSHL<c><q>.I<size> {<Qd>,} <Qm>, #<imm>`	Q = 1 としてエンコードされる
`VSHL<c><q>.I<size> {<Dd>,} <Dm>, #<imm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。ARM `VSHL` は常に無条件命令です。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` L = '0'、imm6〈5:3〉= '001' としてエンコードされます。`<imm>` は、imm6〈2:0〉にエンコードされます。
- `16` L = '0'、imm6〈5:4〉= '01' としてエンコードされます。`<imm>` は、imm6〈3:0〉にエンコードされます。
- `32` L = '0'、imm6〈5〉= '1' としてエンコードされます。`<imm>` は、imm6〈4:0〉にエンコードされます。
- `64` L = '1' としてエンコードされます。`<imm>` は、imm6〈5:0〉にエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<imm>` 0 ～ `<size>` − 1 の範囲のイミディエート値。`<imm>` のエンコード方法については、`<size>` の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = LSL(Elem[D[m+r],e,esize], shift_amount);
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.383 VSHL (register)

Vector Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. (For a rounding shift, see *VRSHL* on page A8-736).

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Encoding T1 / A1 Advanced SIMD

```
VSHL<c>.I<size> <Qd>, <Qm>, <Qn>
VSHL<c>.I<size> <Dd>, <Dm>, <Dn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	0	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	0	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.383 VSHL（レジスタ）

VSHL（ベクタ左シフト（レジスタ））は、ベクタの各エレメントを取得して、2 番目のベクタの対応するエレメントの最下位バイトにより指定されたビット数だけシフトし、結果をデスティネーションベクタに配置します。シフトの値が正の場合、演算は左シフトになります。シフトの値が負の場合、演算は切り捨て付き右シフトになります（丸め付きシフトについては、*VRSHL* P. A8-736 を参照して下さい）。

最初のオペランドと結果のエレメントのデータ型は同じで、次のいずれかを使用できます。

- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号付き整数
- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号なし整数

2 番目のオペランドは、同じサイズの符号付き整数です。

エンコード T1/A1 アドバンスト SIMD

```
VSHL<c>.I<size> <Qd>, <Qm>, <Qn>
VSHL<c>.I<size> <Dd>, <Dm>, <Dn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	1	0	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	1	0	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VSHL<c><q>.<type><size> {<Qd>,} <Qm>, <Qn>`	Encoded as Q = 1
`VSHL<c><q>.<type><size> {<Dd>,} <Dm>, <Dn>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VSHL` instruction must be unconditional.

`<type>` The data type for the elements of the vectors. It must be one of:

- `S` signed, encoded as U = 0
- `U` unsigned, encoded as U = 1.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10
- `64` encoded as size = 0b11.

`<Qd>, <Qm>, <Qn>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dm>, <Dn>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            result = Int(Elem[D[m+r],e,esize], unsigned) << shift;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VSHL<c><q>.<type><size> {<Qd>,} <Qm>, <Qn>`	Q = 1 としてエンコードされる
`VSHL<c><q>.<type><size> {<Dd>,} <Dm>, <Dn>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VSHL` は常に無条件命令です。

`<type>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S` 符号付き、U = 0 としてエンコードされます。
- `U` 符号なし、U = 1 としてエンコードされます。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。
- `64` size = 0b11 としてエンコードされます。

`<Qd>, <Qm>, <Qn>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>, <Dn>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            result = Int(Elem[D[m+r],e,esize], unsigned) << shift;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.384 VSHLL

Vector Shift Left Long takes each element in a doubleword vector, left shifts them by an immediate value, and places the results in a quadword vector.

The operand elements can be:

- 8-bit, 16-bit, or 32-bit signed integers
- 8-bit, 16-bit, or 32-bit unsigned integers
- 8-bit, 16-bit, or 32-bit untyped integers (maximum shift only).

The result elements are twice the length of the operand elements.

Encoding T1 / A1 Advanced SIMD

```
VSHLL<c>.<type><size> <Qd>, <Dm>, #<imm>                    (0 < <imm> < <size>)
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				1	0	1	0	0	0	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				1	0	1	0	0	0	M	1	Vm			

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE VMOVL;
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);
```

Encoding T2 / A2 Advanced SIMD

```
VSHLL<c>.<type><size> <Qd>, <Dm>, #<imm>                    (<imm> == <size>)
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	1	1	0	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	1	1	0	0	M	0	Vm			

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  shift_amount = esize;
unsigned = FALSE;  // Or TRUE without change of functionality
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.384 VSHLL

VSHLL（ベクタ左シフトロング）は、ダブルワードベクタの各エレメントを取得して、イミディエート値分だけ左にシフトし、結果をクワッドワードベクタに配置します。

オペランドのエレメントには、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットの符号付き整数
- 8 ビット、16 ビット、または 32 ビットの符号なし整数
- 8 ビット、16 ビット、または 32 ビットの型なし整数（最大シフトのみ）

結果のエレメントの長さは、オペランドのエレメントの 2 倍です。

エンコード T1/A1 アドバンスト SIMD

```
VSHLL<c>.<type><size> <Qd>, <Dm>, #<imm>                    (0 < <imm> < <size>)
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				1	0	1	0	0	0	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				1	0	1	0	0	0	M	1	Vm			

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE VMOVL;
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);
```

エンコード T2/A2 アドバンスト SIMD

```
VSHLL<c>.<type><size> <Qd>, <Dm>, #<imm>                    (<imm> == <size>)
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	1	1	0	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	1	1	0	0	M	0	Vm			

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  shift_amount = esize;
unsigned = FALSE;  // Or TRUE without change of functionality
d = UInt(D:Vd);  m = UInt(M:Vm);
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VSHLL<c><q>.<type><size>  <Qd>, <Dm>, #<imm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7. An ARM VSHLL instruction must be unconditional.

<type> The data type for the elements of the operand. It must be one of:
- S encoded as U = 0 in encoding T1 / A1
- U encoded as U = 1 in encoding T1 / A1
- I available only in encoding T2 / A2.

<size> The data size for the elements of the operand. It must be one of:
- 8 encoded as imm6<5:3> = '001' or size = '00'
- 16 encoded as imm6<5:4> = '01' or size = '01'
- 32 encoded as imm6<5> = '1' or size = '10'.

<Qd>, <Dm> The destination vector and the operand vector.

<imm> The immediate value. <imm> must lie in the range 1 to <size>:
- if <size> = <imm>, encoding is T2 / A2
- if <size> = 8, <imm> is encoded in imm6<2:0>
- if <size> = 16, <imm> is encoded in imm6<3:0>
- if <size> = 32, <imm> is encoded in imm6<4:0>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Int(Elem[D[m],e,esize], unsigned) << shift_amount;
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VSHLL<c><q>.<type><size>  <Qd>, <Dm>, #<imm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM VSHLL は常に無条件命令です。

<type> オペランドのエレメントのデータ型。次のいずれかを使用できます。
- S エンコード T1/A1 で U = 0 としてエンコードされます。
- U エンコード T1/A1 で U = 1 としてエンコードされます。
- I エンコード T2/A2 でのみ使用できます。

<size> オペランドのエレメントのデータサイズ。次のいずれかを使用できます。
- 8 imm6〈5:3〉 = '001' または size = '00' としてエンコードされます。
- 16 imm6〈5:4〉 = '01' または size = '01' としてエンコードされます。
- 32 imm6〈5〉 = '1' または size = '10' としてエンコードされます。

<Qd>, <Dm> デスティネーションベクタとオペランドベクタ

<imm> イミディエート値。<imm> の範囲は 1 ～ <size> です。
- <size> = <imm> の場合、エンコードは T2/A2 です。
- <size> = 8 の場合、<imm> は imm6〈2:0〉 にエンコードされます。
- <size> = 16 の場合、<imm> は imm6〈3:0〉 にエンコードされます。
- <size> = 32 の場合、<imm> は imm6〈4:0〉 にエンコードされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Int(Elem[D[m],e,esize], unsigned) << shift_amount;
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.385 VSHR

Vector Shift Right takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see *VRSHR* on page A8-738.

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Encoding T1 / A1 Advanced SIMD

```
VSHR<c>.<type><size> <Qd>, <Qm>, #<imm>
VSHR<c>.<type><size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	0	0	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	0	0	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.385 VSHR

VSHR（ベクタ右シフト）は、ベクタの各エレメントを取得して、イミディエート値分だけ右にシフトし、切り捨てられた結果をデスティネーションベクタに配置します。丸められる結果については、*VRSHR* P. A8-738 を参照して下さい。

オペランドと結果のエレメントのサイズは同じで、次のいずれかを使用できます。

- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号付き整数
- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号なし整数

エンコード T1/A1 アドバンスト SIMD

```
VSHR<c>.<type><size> <Qd>, <Qm>, #<imm>
VSHR<c>.<type><size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	0	0	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	0	0	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VSHR<c><q>.<type><size>  {<Qd>,} <Qm>, #<imm>        Encoded as Q = 1
VSHR<c><q>.<type><size>  {<Dd>,} <Dm>, #<imm>        Encoded as Q = 0
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VSHR` instruction must be unconditional.

`<type>` The data type for the elements of the vectors. It must be one of:

- `S` signed, encoded as U = 0
- `U` unsigned, encoded as U = 1.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` Encoded as L = '0', imm6<5:3> = '001'. (8– `<imm>`) is encoded in imm6<2:0>.
- `16` Encoded as L = '0', imm6<5:4> = '01'. (16 – `<imm>`) is encoded in imm6<3:0>.
- `32` Encoded as L = '0', imm6<5> = '1'. (32 – `<imm>`) is encoded in imm6<4:0>.
- `64` Encoded as L = '1'. (64 – `<imm>`) is encoded in imm6<5:0>.

`<Qd>, <Qm>` The destination vector, and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector, and the operand vector, for a doubleword operation.

`<imm>` The immediate value, in the range 1 to `<size>`. See the description of `<size>` for how `<imm>` is encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Exceptions

Undefined Instruction.

Pseudo-instructions

```
VSHR.<type><size>  <Qd>, <Qm>, #0     is a synonym for     VMOV <Qd>, <Qm>
VSHR.<type><size>  <Dd>, <Dm>, #0     is a synonym for     VMOV <Dd>, <Dm>
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
VSHR<c><q>.<type><size>  {<Qd>,} <Qm>, #<imm>        Q = 1 としてエンコードされる
VSHR<c><q>.<type><size>  {<Dd>,} <Dm>, #<imm>        Q = 0 としてエンコードされる
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VSHR` は常に無条件命令です。

`<type>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S` 符号付き、U = 0 としてエンコードされます。
- `U` 符号なし、U = 1 としてエンコードされます。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` L = '0'、imm6〈5:3〉= '001' としてエンコードされます。(8 - `<imm>`) は、imm6〈2:0〉にエンコードされます。
- `16` L = '0'、imm6〈5:4〉= '01' としてエンコードされます。(16 - `<imm>`) は、imm6〈3:0〉にエンコードされます。
- `32` L = '0'、imm6〈5〉= '1' としてエンコードされます。(32 - `<imm>`) は、imm6〈4:0〉にエンコードされます。
- `64` L = '1' としてエンコードされます。(64 - `<imm>`) は、imm6〈5:0〉にエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<imm>` 1 ～ `<size>` の範囲のイミディエート値。`<imm>` のエンコード方法については、`<size>` の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

例外

未定義命令

擬似命令

```
VSHR.<type><size>  <Qd>, <Qm>, #0     は次の命令と同義語     VMOV <Qd>, <Qm>
VSHR.<type><size>  <Dd>, <Dm>, #0     は次の命令と同義語     VMOV <Dd>, <Dm>
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.386 VSHRN

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see *VRSHRN* on page A8-740.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Encoding T1 / A1 Advanced SIMD

```
VSHRN<c>.I<size> <Dd>, <Qm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	imm6						Vd				1	0	0	0	0	0	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	imm6						Vd				1	0	0	0	0	0	M	1	Vm			

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.386 VSHRN

VSHRN（ベクタ右シフト縮小）は、ベクタの各エレメントを取得して、イミディエート値だけ右にシフトし、切り捨てられた結果をデスティネーションベクタに配置します。丸められる結果については、*VRSHRN* P. A8-740 を参照して下さい。

オペランドのエレメントには、16 ビット、32 ビット、または 64 ビットの整数を使用できます。整数には符号付きと符号なしの区別はありません。デスティネーションのエレメントのサイズは、ソースのエレメントの半分です。

エンコード T1/A1 アドバンスト SIMD

```
VSHRN<c>.I<size> <Dd>, <Qm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	imm6						Vd				1	0	0	0	0	0	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	imm6						Vd				1	0	0	0	0	0	M	1	Vm			

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VSHRN<c><q>.I<size>  <Dd>, <Qm>, #<imm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7. An ARM VSHRN instruction must be unconditional.

<size> The data size for the elements of the vectors. It must be one of:

- 16 Encoded as imm6<5:3> = '001'. (8 – <imm>) is encoded in imm6<2:0>.
- 32 Encoded as imm6<5:4> = '01'. (16 – <imm>) is encoded in imm6<3:0>.
- 64 Encoded as imm6<5> = '1'. (32 – <imm>) is encoded in imm6<4:0>.

<Dd>, <Qm> The destination vector, and the operand vector.

<imm> The immediate value, in the range 1 to <size>/2. See the description of <size> for how <imm> is encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = LSR(Elem[Q[m>>1],e,2*esize], shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```

Exceptions

Undefined Instruction.

Pseudo-instructions

VSHRN.I<size> <Dd>, <Qm>, #0 is a synonym for VMOVN.I<size> <Dd>, <Qm>

For details see *VMOVN* on page A8-656.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VSHRN<c><q>.I<size>  <Dd>, <Qm>, #<imm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM VSHRN は常に無条件命令です。

<size> ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- 16 imm6〈5:3〉= '001' としてエンコードされます。(8 - <imm>) は、imm6〈2:0〉にエンコードされます。
- 32 imm6〈5:4〉= '01' としてエンコードされます。(16 - <imm>) は、imm6〈3:0〉にエンコードされます。
- 64 imm6〈5〉= '1' としてエンコードされます。(32 - <imm>) は、imm6〈4:0〉にエンコードされます。

<Dd>, <Qm> デスティネーションベクタとオペランドベクタ

<imm> 1 ～ <size> / 2 の範囲のイミディエート値。<imm> のエンコード方法については、<size> の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = LSR(Elem[Q[m>>1],e,2*esize], shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```

例外

未定義命令

擬似命令

VSHRN.I<size> <Dd>, <Qm>, #0 は次の命令と同義語VMOVN.I<size> <Dd>, <Qm>

詳細については、*VMOVN* P. A8-656 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.387 VSLI

Vector Shift Left and Insert takes each element in the operand vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Encoding T1 / A1 Advanced SIMD

```
VSLI<c>.<size> <Qd>, <Qm>, #<imm>
VSLI<c>.<size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.387 VSLI

VSLI（ベクタ左シフトおよび挿入）は、オペランドベクタの各エレメントを取得して、イミディエート値だけ左にシフトし、結果をデスティネーションベクタに挿入します。各エレメントの左からシフトアウトされたビットは失われます。

エレメントはすべて同じサイズで、8 ビット、16 ビット、32 ビット、または 64 ビットを使用できます。これらにはデータ型の区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VSLI<c>.<size> <Qd>, <Qm>, #<imm>
VSLI<c>.<size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VSLI<c><q>.<size> {<Qd>,} <Qm>, #<imm>`	Encoded as Q = 1
`VSLI<c><q>.<size> {<Dd>,} <Dm>, #<imm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VSLI` instruction must be unconditional.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` Encoded as L = '0', imm6<5:3> = '001'. `<imm>` is encoded in imm6<2:0>.
- `16` Encoded as L = '0', imm6<5:4> = '01'. `<imm>` is encoded in imm6<3:0>.
- `32` Encoded as L = '0', imm6<5> = '1'. `<imm>` is encoded in imm6<4:0>.
- `64` Encoded as L = '1'. `<imm>` is encoded in imm6<5:0>.

`<Qd>, <Qm>` The destination vector, and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector, and the operand vector, for a doubleword operation.

`<imm>` The immediate value, in the range 0 to `<size>`-1. See the description of `<size>` for how `<imm>` is encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    mask = LSL(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            shifted_op = LSL(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VSLI<c><q>.<size> {<Qd>,} <Qm>, #<imm>`	Q = 1 としてエンコードされる
`VSLI<c><q>.<size> {<Dd>,} <Dm>, #<imm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VSLI` は常に無条件命令です。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` L = '0'、imm6〈5:3〉= '001' としてエンコードされます。`<imm>` は、imm6〈2:0〉にエンコードされます。
- `16` L = '0'、imm6〈5:4〉= '01' としてエンコードされます。`<imm>` は、imm6〈3:0〉にエンコードされます。
- `32` L = '0'、imm6〈5〉= '1' としてエンコードされます。`<imm>` は、imm6〈4:0〉にエンコードされます。
- `64` L = '1' としてエンコードされます。`<imm>` は、imm6〈5:0〉にエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<imm>` 0 ～ `<size>` − 1 の範囲のイミディエート値。`<imm>` のエンコード方法については、`<size>` の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    mask = LSL(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            shifted_op = LSL(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.388 VSQRT

This instruction calculates the square root of the value in a floating-point register and writes the result to another floating-point register.

Encoding T1 / A1 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VSQRT<c>.F64 <Dd>, <Dm>
VSQRT<c>.F32 <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	1	Vd				1	0	1	sz	1	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	0	0	1	Vd				1	0	1	sz	1	1	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP vectors This instruction can operate on VFP vectors under control of the FPSCR.LEN and FPSCR.STRIDE bits. For details see Appendix F *VFP Vector Operation Support*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.388 VSQRT

VSQRT は、浮動小数点レジスタの値の平方根を計算し、結果を別の浮動小数点レジスタに書き込みます。

エンコード T1/A1 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VSQRT<c>.F64 <Dd>, <Dm>
VSQRT<c>.F32 <Sd>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	1	Vd				1	0	1	sz	1	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	D	1	1	0	0	0	1	Vd				1	0	1	sz	1	1	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP ベクタ この命令は、FPSCR.LEN ビットと FPSCR.STRIDE ビットの制御の下で VFP ベクタに作用することができます。詳細については、付録 F *「VFP ベクタ演算のサポート」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VSQRT<c><q>.F64 <Dd>, <Dm>`	Encoded as sz = 1
`VSQRT<c><q>.F32 <Sd>, <Sm>`	Encoded as sz = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Dd>, <Dm>` The destination vector and the operand vector, for a double-precision operation.

`<Sd>, <Sm>` The destination vector and the operand vector, for a single-precision operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        D[d] = FPSqrt(D[m]);
    else
        S[d] = FPSqrt(S[m]);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Invalid Operation, Inexact, Input Denormal.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VSQRT<c><q>.F64 <Dd>, <Dm>`	sz = 1 としてエンコードされる
`VSQRT<c><q>.F32 <Sd>, <Sm>`	sz = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Dd>, <Dm>` 倍精度演算の場合のデスティネーションベクタとオペランドベクタ

`<Sd>, <Sm>` 単精度演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if dp_operation then
        D[d] = FPSqrt(D[m]);
    else
        S[d] = FPSqrt(S[m]);
```

例外

未定義命令

浮動小数点例外：無効演算、不正確、入力非正規化

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.389 VSRA

Vector Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the truncated results into the destination vector. (For rounded results, see *VRSRA* on page A8-746.)

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Encoding T1 / A1 Advanced SIMD

```
VSRA<c>.<type><size> <Qd>, <Qm>, #<imm>
VSRA<c>.<type><size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	0	1	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	0	1	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.389 VSRA

VSRA（ベクタ右シフトおよび累積）は、ベクタの各エレメントを取得して、イミディエート値分だけ右にシフトし、切り捨てられた結果をデスティネーションベクタに累積します（丸められる結果については、*VRSRA* P. A8-746 を参照して下さい）。

オペランドと結果のエレメントのデータ型はすべて同じで、次のいずれかを使用できます。

- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号付き整数
- 8 ビット、16 ビット、32 ビット、または 64 ビットの符号なし整数

エンコード T1/A1 アドバンスト SIMD

```
VSRA<c>.<type><size> <Qd>, <Qm>, #<imm>
VSRA<c>.<type><size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	0	1	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	0	1	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VSRA<c><q>.<type><size> {<Qd>,} <Qm>, #<imm>`	Encoded as Q = 1
`VSRA<c><q>.<type><size> {<Dd>,} <Dm>, #<imm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VSRA` instruction must be unconditional.

`<type>` The data type for the elements of the vectors. It must be one of:

- `S` signed, encoded as U = 0
- `U` unsigned, encoded as U = 1.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` Encoded as L = '0', imm6<5:3> = '001'. (8– `<imm>`) is encoded in imm6<2:0>.
- `16` Encoded as L = '0', imm6<5:4> = '01'. (16 – `<imm>`) is encoded in imm6<3:0>.
- `32` Encoded as L = '0', imm6<5> = '1'. (32 – `<imm>`) is encoded in imm6<4:0>.
- `64` Encoded as L = '1'. (64 – `<imm>`) is encoded in imm6<5:0>.

`<Qd>, <Qm>` The destination vector, and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector, and the operand vector, for a doubleword operation.

`<imm>` The immediate value, in the range 1 to `<size>`. See the description of `<size>` for how `<imm>` is encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VSRA<c><q>.<type><size> {<Qd>,} <Qm>, #<imm>`	Q = 1 としてエンコードされる
`VSRA<c><q>.<type><size> {<Dd>,} <Dm>, #<imm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。ARM `VSRA` は常に無条件命令です。

`<type>` ベクタのエレメントのデータ型。次のいずれかを使用できます。

- `S` 符号付き、U = 0 としてエンコードされます。
- `U` 符号なし、U = 1 としてエンコードされます。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` L = '0'、imm6〈5:3〉= '001' としてエンコードされます。(8 - `<imm>`) は、imm6〈2:0〉にエンコードされます。
- `16` L = '0'、imm6〈5:4〉= '01' としてエンコードされます。(16 - `<imm>`) は、imm6〈3:0〉にエンコードされます。
- `32` L = '0'、imm6〈5〉= '1' としてエンコードされます。(32 - `<imm>`) は、imm6〈4:0〉にエンコードされます。
- `64` L = '1' としてエンコードされます。(64 - `<imm>`) は、imm6〈5:0〉にエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<imm>` 1 ～ `<size>` の範囲のイミディエート値。`<imm>` のエンコード方法については、`<size>` の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.390 VSRI

Vector Shift Right and Insert takes each element in the operand vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Encoding T1 / A1 Advanced SIMD

```
VSRI<c>.<size> <Qd>, <Qm>, #<imm>
VSRI<c>.<size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	imm6						Vd				0	1	0	0	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	imm6						Vd				0	1	0	0	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings See *One register and a modified immediate value* on page A7-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.390 VSRI

VSRI（ベクタ右シフトおよび挿入）は、オペランドベクタの各エレメントを取得して、イミディエート値分だけ右にシフトし、結果をデスティネーションベクタに挿入します。各エレメントの右端からシフトアウトされたビットは失われます。

エレメントはすべて同じサイズで、8 ビット、16 ビット、32 ビット、または 64 ビットを使用できます。これらにはデータ型の区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VSRI<c>.<size> <Qd>, <Qm>, #<imm>
VSRI<c>.<size> <Dd>, <Dm>, #<imm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	imm6						Vd				0	1	0	0	L	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	imm6						Vd				0	1	0	0	L	Q	M	1	Vm			

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

関連エンコード *1 つのレジスタと修飾イミディエート値* P. A7-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VSRI<c><q>.<size> {<Qd>,} <Qm>, #<imm>`	Encoded as Q = 1
`VSRI<c><q>.<size> {<Dd>,} <Dm>, #<imm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VSRI` instruction must be unconditional.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` Encoded as L = '0', imm6<5:3> = '001'. (8– `<imm>`) is encoded in imm6<2:0>.
- `16` Encoded as L = '0', imm6<5:4> = '01'. (16 – `<imm>`) is encoded in imm6<3:0>.
- `32` Encoded as L = '0', imm6<5> = '1'. (32 – `<imm>`) is encoded in imm6<4:0>.
- `64` Encoded as L = '1'. (64 – `<imm>`) is encoded in imm6<5:0>.

`<Qd>, <Qm>` The destination vector, and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector, and the operand vector, for a doubleword operation.

`<imm>` The immediate value, in the range 1 to `<size>`. See the description of `<size>` for how `<imm>` is encoded.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    mask = LSR(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            shifted_op = LSR(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VSRI<c><q>.<size> {<Qd>,} <Qm>, #<imm>`	Q = 1 としてエンコードされる
`VSRI<c><q>.<size> {<Dd>,} <Dm>, #<imm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。ARM `VSRI` は常に無条件命令です。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` L = '0'、imm6〈5:3〉= '001' としてエンコードされます。(8 - `<imm>`) は、imm6〈2:0〉にエンコードされます。
- `16` L = '0'、imm6〈5:4〉= '01' としてエンコードされます。(16 - `<imm>`) は、imm6〈3:0〉にエンコードされます。
- `32` L = '0'、imm6〈5〉= '1' としてエンコードされます。(32 - `<imm>`) は、imm6〈4:0〉にエンコードされます。
- `64` L = '1' としてエンコードされます。(64 - `<imm>`) は、imm6〈5:0〉にエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

`<imm>` 1 ～ `<size>` の範囲のイミディエート値。`<imm>` のエンコード方法については、`<size>` の説明を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    mask = LSR(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            shifted_op = LSR(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.391 VST1 (multiple single elements)

Vector Store (multiple single elements) stores elements to memory from one, two, three, or four registers, without interleaving. Every element of each register is stored. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VST1<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST1<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				type				size		align		Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				type				size		align		Rm			

```
case type of
    when '0111'
        regs = 1;  if align<1> == '1' then UNDEFINED;
    when '1010'
        regs = 2;  if align == '11' then UNDEFINED;
    when '0110'
        regs = 3;  if align<1> == '1' then UNDEFINED;
    when '0010'
        regs = 4;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d+regs > 32 then UNPREDICTABLE;
```

Related encodings See *Advanced SIMD element or structure load/store instructions* on page A7-27

Assembler syntax

```
VST1<c><q>.<size>  <list>, [<Rn>{@<align>}]          Rm = '1111'
VST1<c><q>.<size>  <list>, [<Rn>{@<align>}]!         Rm = '1101'
VST1<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>    Rm = other values
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VST1` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.391 VST1（複数の単一エレメントのストア）

VST1（ベクタストア（複数の単一エレメントのストア））は、エレメントを 1 つ、2 つ、3 つ、または 4 つのレジスタからメモリに、インターリーブなしでストアします。各レジスタのすべてのエレメントがストアされます。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VST1<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST1<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				type				size		align		Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				type				size		align		Rm			

```
case type of
    when '0111'
        regs = 1;  if align<1> == '1' then UNDEFINED;
    when '1010'
        regs = 2;  if align == '11' then UNDEFINED;
    when '0110'
        regs = 3;  if align<1> == '1' then UNDEFINED;
    when '0010'
        regs = 4;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d+regs > 32 then UNPREDICTABLE;
```

関連エンコード *アドバンスト SIMD のエレメントまたは構造ロード / ストア命令* P. A7-27 を参照して下さい。

アセンブラ構文

```
VST1<c><q>.<size>  <list>, [<Rn>{@<align>}]          Rm = '1111'
VST1<c><q>.<size>  <list>, [<Rn>{@<align>}]!         Rm = '1101'
VST1<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>    Rm = その他の値
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VST1` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

32 encoded as size = 0b10

64 encoded as size = 0b11.

<list> The list of registers to store. It must be one of:

{<Dd>} encoded as D:Vd = <Dd>, type = 0b0111

{<Dd>, <Dd+1>} encoded as D:Vd = <Dd>, type = 0b1010

{<Dd>, <Dd+1>, <Dd+2>}
encoded as D:Vd = <Dd>, type = 0b0110

{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>}
encoded as D:Vd = <Dd>, type = 0b0010.

<Rn> Contains the base address for the access.

<align> The alignment. It can be one of:

64 8-byte alignment, encoded as align = 0b01.

128 16-byte alignment, available only if <list> contains two or four registers, encoded as align = 0b10.

256 32-byte alignment, available only if <list> contains four registers, encoded as align = 0b11.

omitted Standard alignment, see *Unaligned data access* on page A3-5. Encoded as align = 0b00.

! If present, specifies writeback.

<Rm> Contains an address offset applied after the access.

For more information about <Rn>, !, and <Rm>, see *Advanced SIMD addressing mode* on page A7-30.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 8*regs);
    for r = 0 to regs-1
        for e = 0 to elements-1
            MemU[address,ebytes] = Elem[D[d+r],e,esize];
            address = address + ebytes;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

16 size = 0b01 としてエンコードされます。

32 size = 0b10 としてエンコードされます。

64 size = 0b11 としてエンコードされます。

<list> ストアするレジスタのリスト。次のいずれかを使用できます。

{<Dd>} D:Vd = <Dd>、type = 0b0111 としてエンコードされます。

{<Dd>, <Dd+1>} D:Vd = <Dd>、type = 0b1010 としてエンコードされます。

{<Dd>, <Dd+1>, <Dd+2>}
D:Vd = <Dd>、type = 0b0110 としてエンコードされます。

{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>}
D:Vd = <Dd>、type = 0b0010 としてエンコードされます。

<Rn> アクセス用のベースアドレス。

<align> アライメント。次のいずれかを使用できます。

64 8 バイトアライメント。align = 0b01 としてエンコードされます。

128 16 バイトアライメント。<list> に 2 つまたは 4 つのレジスタが含まれる場合のみ使用できます。align = 0b10 としてエンコードされます。

256 32 バイトアライメント。<list> に 4 つのレジスタが含まれる場合のみ使用できます。align = 0b11 としてエンコードされます。

省略 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。align = 0b00 としてエンコードされます。

! 存在する場合、ライトバックが指定されます。

<Rm> アクセス後に適用されるアドレスオフセット

<Rn>、!、および <Rm> の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 8*regs);
    for r = 0 to regs-1
        for e = 0 to elements-1
            MemU[address,ebytes] = Elem[D[d+r],e,esize];
            address = address + ebytes;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.392 VST1 (single element from one lane)

This instruction stores one element to memory from one element of a register. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VST1<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST1<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				size		0	0	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				size		0	0	index_align				Rm			

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  alignment = 1;
    when '01'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        alignment = if index_align<0> == '0' then 1 else 2;
    when '10'
        if index_align<2> != '0' then UNDEFINED;
        if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
```

Assembler syntax

`VST1<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VST1<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VST1<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VST1` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.392 VST1（1 つのレーンからの単一エレメントのストア）

VST1（1 つのレーンからの単一エレメントのストア）は、1 つのエレメントをレジスタの 1 つのエレメントからメモリにストアします。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VST1<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST1<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				size		0	0	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				size		0	0	index_align				Rm			

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  alignment = 1;
    when '01'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        alignment = if index_align<0> == '0' then 1 else 2;
    when '10'
        if index_align<2> != '0' then UNDEFINED;
        if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
```

アセンブラ構文

`VST1<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VST1<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VST1<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VST1` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` The register containing the element to store. It must be {`<Dd[x]>`}. The register `Dd` is encoded in D:Vd

`<Rn>` Contains the base address for the access.

`<align>` The alignment. It can be one of:

- `16` 2-byte alignment, available only if `<size>` is 16
- `32` 4-byte alignment, available only if `<size>` is 32
- **omitted** Standard alignment, see *Unaligned data access* on page A3-5.

`!` If present, specifies writeback.

`<Rm>` Contains an address offset applied after the access.

For more information about `<Rn>`, `!`, and `<Rm>`, see *Advanced SIMD addressing mode* on page A7-30.

Table A8-9 shows the encoding of index and alignment for different `<size>` values.

Table A8-9 Encoding of index and alignment

	`<size>` == 8	`<size>` == 16	`<size>` == 32
Index	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
`<align>` omitted	index_align[0] = 0	index_align[1:0] = ’00’	index_align[2:0] = ’000’
`<align>` == 16	-	index_align[1:0] = ’01’	-
`<align>` == 32	-	-	index_align[2:0] = ’011’

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else ebytes);
    MemU[address,ebytes] = Elem[D[d],index,esize];
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` ストアするエレメントを含むレジスタ。{`<Dd[x]>`} にする必要があります。レジスタ `Dd` は、D:Vd にエンコードされます。

`<Rn>` アクセス用のベースアドレス。

`<align>` アライメント。次のいずれかを使用できます。

- `16` 2 バイトアライメント。`<size>` が 16 の場合にのみ使用できます。
- `32` 4 バイトアライメント。`<size>` が 32 の場合にのみ使用できます。
- **省略** 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。

`!` 存在する場合、ライトバックが指定されます。

`<Rm>` アクセス後に適用されるアドレスオフセット

`<Rn>`、`!`、および `<Rm>` の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

さまざまな `<size>` 値でのインデクスとアライメントのエンコードを表 A8-9 に示します。

表 A8-9 インデクスとアライメントのエンコード

	`<size>` == 8	`<size>` == 16	`<size>` == 32
インデクス	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
`<align>` 省略	index_align[0] = 0	index_align[1:0] = ’00’	index_align[2:0] = ’000’
`<align>` == 16	–	index_align[1:0] = ’01’	–
`<align>` == 32	–	–	index_align[2:0] = ’011’

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else ebytes);
    MemU[address,ebytes] = Elem[D[d],index,esize];
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.393 VST2 (multiple 2-element structures)

This instruction stores multiple 2-element structures from two or four registers to memory, with interleaving. For more information, see *Element and structure load/store instructions* on page A4-27. Every element of each register is saved. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VST2<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST2<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				type				size		align		Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				type				size		align		Rm			

```
if size == '11' then UNDEFINED;
case type of
    when '1000'
        regs = 1;  inc = 1;  if align == '11' then UNDEFINED;
    when '1001'
        regs = 1;  inc = 2;  if align == '11' then UNDEFINED;
    when '0011'
        regs = 2;  inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d2+regs > 32 then UNPREDICTABLE;
```

Related encodings See *Advanced SIMD element or structure load/store instructions* on page A7-27

Assembler syntax

`VST2<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VST2<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VST2<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VST2` instruction must be unconditional.

`<size>` The data size. It must be one of:

- 8 encoded as size = 0b00
- 16 encoded as size = 0b01

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.393 VST2（複数の 2 エレメント構造のストア）

VST2（複数の 2 エレメント構造のストア）は、複数の 2 エレメント構造を 2 つまたは 4 つのレジスタからメモリに、インターリーブ付きでストアします。詳細については、*エレメントおよび構造のロード / ストア命令* P. A4-27 を参照して下さい。各レジスタのすべてのエレメントが保存されます。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VST2<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST2<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				type				size		align		Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				type				size		align		Rm			

```
if size == '11' then UNDEFINED;
case type of
    when '1000'
        regs = 1;  inc = 1;  if align == '11' then UNDEFINED;
    when '1001'
        regs = 1;  inc = 2;  if align == '11' then UNDEFINED;
    when '0011'
        regs = 2;  inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d2+regs > 32 then UNPREDICTABLE;
```

関連エンコード *アドバンスト SIMD のエレメントまたは構造ロード / ストア命令* P. A7-27 を参照して下さい。

アセンブラ構文

`VST2<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VST2<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VST2<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VST2` は常に無条件命令です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

32 encoded as size = 0b10.

<list> The list of registers to store. It must be one of:

{<Dd>, <Dd+1>} encoded as D:Vd = <Dd>, type = 0b1000

{<Dd>, <Dd+2>} encoded as D:Vd = <Dd>, type = 0b1001

{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>}
encoded as D:Vd = <Dd>, type = 0b0011.

<Rn> Contains the base address for the access.

<align> The alignment. It can be one of:

64 8-byte alignment, encoded as align = 0b01.

128 16-byte alignment, encoded as align = 0b10.

256 32-byte alignment, available only if <list> contains four registers, encoded as align = 0b11

omitted Standard alignment, see *Unaligned data access* on page A3-5. Encoded as align = 0b00.

! If present, specifies writeback.

<Rm> Contains an address offset applied after the access.

For more information about <Rn>, !, and <Rm>, see *Advanced SIMD addressing mode* on page A7-30.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 16*regs);
    for r = 0 to regs-1
        for e = 0 to elements-1
            MemU[address,ebytes] = Elem[D[d+r],e,esize];
            MemU[address+ebytes,ebytes] = Elem[D[d2+r],e,esize];
            address = address + 2*ebytes;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<size> データサイズ。次のいずれかを使用できます。

8 size = 0b00 としてエンコードされます。

16 size = 0b01 としてエンコードされます。

32 size = 0b10 としてエンコードされます。

<list> ストアするレジスタのリスト。次のいずれかを使用できます。

{<Dd>, <Dd+1>} D:Vd = <Dd>、type = 0b1000 としてエンコードされます。

{<Dd>, <Dd+2>} D:Vd = <Dd>、type = 0b1001 としてエンコードされます。

{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>}
D:Vd = <Dd>、type = 0b0011 としてエンコードされます。

<Rn> アクセス用のベースアドレス。

<align> アライメント。次のいずれかを使用できます。

64 8 バイトアライメント。align = 0b01 としてエンコードされます。

128 16 バイトアライメント。align = 0b10 としてエンコードされます。

256 32 バイトアライメント。<list> に 4 つのレジスタが含まれる場合のみ使用できます。align = 0b11 としてエンコードされます。

省略 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。align = 0b00 としてエンコードされます。

! 存在する場合、ライトバックが指定されます。

<Rm> アクセス後に適用されるアドレスオフセット

<Rn>、!、および <Rm> の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 16*regs);
    for r = 0 to regs-1
        for e = 0 to elements-1
            MemU[address,ebytes] = Elem[D[d+r],e,esize];
            MemU[address+ebytes,ebytes] = Elem[D[d2+r],e,esize];
            address = address + 2*ebytes;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.394 VST2 (single 2-element structure from one lane)

This instruction stores one 2-element structure to memory from corresponding elements of two registers. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VST2<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST2<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				size		0	1	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				size		0	1	index_align				Rm			

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 2;
    when '01'
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '10'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d2 > 31 then UNPREDICTABLE;
```

Assembler syntax

`VST2<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VST2<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VST2<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VST2` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.394 VST2（1 つのレーンからの単一の 2 エレメント構造のストア）

VST2（1 つのレーンからの単一の 2 エレメント構造のストア）は、1 つの 2 エレメント構造を、2 つのレジスタの対応するエレメントからメモリにストアします。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VST2<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST2<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				size		0	1	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				size		0	1	index_align				Rm			

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 2;
    when '01'
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '10'
        if index_align<1> != '0' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d2 > 31 then UNPREDICTABLE;
```

アセンブラ構文

`VST2<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VST2<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VST2<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VST2` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` The registers containing the structure. Encoded with D:Vd = `<Dd>`. It must be one of:

- `{<Dd[x]>, <Dd+1[x]>}` Single-spaced registers, see Table A8-10.
- `{<Dd[x]>, <Dd+2[x]>}` Double-spaced registers, see Table A8-10. This is not available if `<size>` == 8.

`<Rn>` Contains the base address for the access.

`<align>` The alignment. It can be one of:

- `16` 2-byte alignment, available only if `<size>` is 8
- `32` 4-byte alignment, available only if `<size>` is 16
- `64` 8-byte alignment, available only if `<size>` is 32
- **omitted** Standard alignment, see *Unaligned data access* on page A3-5.

`!` If present, specifies writeback.

`<Rm>` Contains an address offset applied after the access.

For more information about `<Rn>`, `!`, and `<Rm>`, see *Advanced SIMD addressing mode* on page A7-30.

Table A8-10 Encoding of index, alignment, and register spacing

	`<size>` == 8	`<size>` == 16	`<size>` == 32
Index	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
Single-spacing	-	index_align[1] = 0	index_align[2] = 0
Double-spacing	-	index_align[1] = 1	index_align[2] = 1
`<align>` omitted	index_align[0] = 0	index_align[0] = 0	index_align[1:0] = '00'
`<align>` == 16	index_align[0] = 1	-	-
`<align>` == 32	-	index_align[0] = 1	-
`<align>` == 64	-	-	index_align[1:0] = '01'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 2*ebytes);
    MemU[address,ebytes] = Elem[D[d],index,esize];
    MemU[address+ebytes,ebytes] = Elem[D[d2],index,esize];
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` 構造を含むレジスタ。D:Vd = `<Dd>` としてエンコードされます。次のいずれかを使用できます。

- `{<Dd[x]>, <Dd+1[x]>}` シングルスペース レジスタ（表 A8-10 参照）。
- `{<Dd[x]>, <Dd+2[x]>}` ダブルスペース レジスタ（表 A8-10 参照）。`<size>` == 8 の場合には使用できません。

`<Rn>` アクセス用のベースアドレス。

`<align>` アライメント。次のいずれかを使用できます。

- `16` 2 バイトアライメント。`<size>` が 8 の場合にのみ使用できます。
- `32` 4 バイトアライメント。`<size>` が 16 の場合にのみ使用できます。
- `64` 8 バイトアライメント。`<size>` が 32 の場合にのみ使用できます。
- **省略** 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。

`!` 存在する場合、ライトバックが指定されます。

`<Rm>` アクセス後に適用されるアドレスオフセット

`<Rn>`、`!`、および `<Rm>` の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

表 A8-10 インデクス、アライメント、およびレジスタスペーシングのエンコード

	`<size>` == 8	`<size>` == 16	`<size>` == 32
インデクス	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
シングルスペーシング	–	index_align[1] = 0	index_align[2] = 0
ダブルスペーシング	–	index_align[1] = 1	index_align[2] = 1
`<align>` 省略	index_align[0] = 0	index_align[0] = 0	index_align[1:0] = '00'
`<align>` == 16	index_align[0] = 1	–	–
`<align>` == 32	–	index_align[0] = 1	–
`<align>` == 64	–	–	index_align[1:0] = '01'

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 2*ebytes);
    MemU[address,ebytes] = Elem[D[d],index,esize];
    MemU[address+ebytes,ebytes] = Elem[D[d2],index,esize];
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.395 VST3 (multiple 3-element structures)

This instruction stores multiple 3-element structures to memory from three registers, with interleaving. For more information, see *Element and structure load/store instructions* on page A4-27. Every element of each register is saved. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VST3<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST3<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				type				size		align		Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				type				size		align		Rm			

```
if size == '11' || align<1> == '1' then UNDEFINED;
case type of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d3 > 31 then UNPREDICTABLE;
```

Related encodings See *Advanced SIMD element or structure load/store instructions* on page A7-27

Assembler syntax

```
VST3<c><q>.<size>  <list>, [<Rn>{@<align>}]          Rm = '1111'
VST3<c><q>.<size>  <list>, [<Rn>{@<align>}]!         Rm = '1101'
VST3<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>    Rm = other values
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VST3` instruction must be unconditional.

`<size>` The data size. It must be one of:

- 8 encoded as size = 0b00
- 16 encoded as size = 0b01
- 32 encoded as size = 0b10.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.395 VST3（複数の 3 エレメント構造のストア）

VST3（複数の 3 エレメント構造のストア）は、複数の 3 エレメント構造を 3 つのレジスタからメモリに、インターリーブ付きでストアします。詳細については、*エレメントおよび構造のロード／ストア命令* P. A4-27 を参照して下さい。各レジスタのすべてのエレメントが保存されます。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VST3<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST3<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				type				size		align		Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				type				size		align		Rm			

```
if size == '11' || align<1> == '1' then UNDEFINED;
case type of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d3 > 31 then UNPREDICTABLE;
```

関連エンコード *アドバンスト SIMD のエレメントまたは構造ロード／ストア命令* P. A7-27 を参照して下さい。

アセンブラ構文

```
VST3<c><q>.<size>  <list>, [<Rn>{@<align>}]          Rm = '1111'
VST3<c><q>.<size>  <list>, [<Rn>{@<align>}]!         Rm = '1101'
VST3<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>    Rm = その他の値
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VST3` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- 8 size = 0b00 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` The list of registers to store. It must be one of:

`{<Dd>, <Dd+1>, <Dd+2>}`
encoded as D:Vd = `<Dd>`, type = 0b0100

`{<Dd>, <Dd+2>, <Dd+4>}`
encoded as D:Vd = `<Dd>`, type = 0b0101.

`<Rn>` Contains the base address for the access.

`<align>` The alignment. It can be:

`64` 8-byte alignment, encoded as align = 0b01.

omitted Standard alignment, see *Unaligned data access* on page A3-5. Encoded as align = 0b00.

`!` If present, specifies writeback.

`<Rm>` Contains an address offset applied after the access.

For more information about `<Rn>`, `!`, and `<Rm>`, see *Advanced SIMD addressing mode* on page A7-30.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 24);
    for e = 0 to elements-1
        MemU[address,ebytes] = Elem[D[d],e,esize];
        MemU[address+ebytes,ebytes] = Elem[D[d2],e,esize];
        MemU[address+2*ebytes,ebytes] = Elem[D[d3],e,esize];
        address = address + 3*ebytes;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`16` size = 0b01 としてエンコードされます。

`32` size = 0b10 としてエンコードされます。

`<list>` ストアするレジスタのリスト。次のいずれかを使用できます。

`{<Dd>, <Dd+1>, <Dd+2>}`
D:Vd = `<Dd>`、type = 0b0100 としてエンコードされます。

`{<Dd>, <Dd+2>, <Dd+4>}`
D:Vd = `<Dd>`、type = 0b0101 としてエンコードされます。

`<Rn>` アクセス用のベースアドレス。

`<align>` アライメント。次のいずれかを使用できます。

`64` 8 バイトアライメント。align = 0b01 としてエンコードされます。

省略 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。align = 0b00 としてエンコードされます。

`!` 存在する場合、ライトバックが指定されます。

`<Rm>` アクセス後に適用されるアドレスオフセット

`<Rn>`、`!`、および `<Rm>` の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 24);
    for e = 0 to elements-1
        MemU[address,ebytes] = Elem[D[d],e,esize];
        MemU[address+ebytes,ebytes] = Elem[D[d2],e,esize];
        MemU[address+2*ebytes,ebytes] = Elem[D[d3],e,esize];
        address = address + 3*ebytes;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.396 VST3 (single 3-element structure from one lane)

This instruction stores one 3-element structure to memory from corresponding elements of three registers. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VST3<c>.<size> <list>, [<Rn>]{!}
VST3<c>.<size> <list>, [<Rn>], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				size		1	0	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				size		1	0	index_align				Rm			

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  inc = 1;
    when '01'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
    when '10'
        if index_align<1:0> != '00' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d3 > 31 then UNPREDICTABLE;
```

Assembler syntax

`VST3<c><q>.<size> <list>, [<Rn>]`	Rm = '1111'
`VST3<c><q>.<size> <list>, [<Rn>]!`	Rm = '1101'
`VST3<c><q>.<size> <list>, [<Rn>], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VST3` instruction must be unconditional.

`<size>` The data size. It must be one of:

- 8 encoded as size = 0b00
- 16 encoded as size = 0b01
- 32 encoded as size = 0b10.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.396 VST3（1 つのレーンからの単一の 3 エレメント構造のストア）

VST3（1 つのレーンからの単一の 3 エレメント構造のストア）は、1 つの 3 エレメント構造を、3 つのレジスタの対応するエレメントからメモリにストアします。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VST3<c>.<size> <list>, [<Rn>]{!}
VST3<c>.<size> <list>, [<Rn>], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				size		1	0	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				size		1	0	index_align				Rm			

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  inc = 1;
    when '01'
        if index_align<0> != '0' then UNDEFINED;
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
    when '10'
        if index_align<1:0> != '00' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d3 > 31 then UNPREDICTABLE;
```

アセンブラ構文

`VST3<c><q>.<size> <list>, [<Rn>]`	Rm = '1111'
`VST3<c><q>.<size> <list>, [<Rn>]!`	Rm = '1101'
`VST3<c><q>.<size> <list>, [<Rn>], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VST3` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- 8 size = 0b00 としてエンコードされます。
- 16 size = 0b01 としてエンコードされます。
- 32 size = 0b10 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` The registers containing the structure. Encoded with D:Vd = `<Dd>`. It must be one of:

`{<Dd[x]>, <Dd+1[x]>, <Dd+2[x]>}`
Single-spaced registers, see Table A8-11.

`{<Dd[x]>, <Dd+2[x]>, <Dd+4[x]>}`
Double-spaced registers, see Table A8-11. This is not available if `<size>` == 8.

`<Rn>` Contains the base address for the access.

`!` If present, specifies writeback.

`<Rm>` Contains an address offset applied after the access.

For more information about `<Rn>`, `!`, and `<Rm>`, see *Advanced SIMD addressing mode* on page A7-30.

Table A8-11 Encoding of index and register spacing

	`<size>` == 8	`<size>` == 16	`<size>` == 32
Index	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
Single-spacing	index_align[0] = 0	index_align[1:0] = ’00’	index_align[2:0] = ’000’
Double-spacing	-	index_align[1:0] = ’10’	index_align[2:0] = ’100’

Alignment

Standard alignment rules apply, see *Unaligned data access* on page A3-5.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];
    if wback then R[n] = R[n] + (if register_index then R[m] else 3*ebytes);
    MemU[address,ebytes] = Elem[D[d],index,esize];
    MemU[address+ebytes,ebytes] = Elem[D[d2],index,esize];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index,esize];
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` 構造を含むレジスタ。D:Vd = `<Dd>` としてエンコードされます。次のいずれかを使用できます。

`{<Dd[x]>, <Dd+1[x]>, <Dd+2[x]>}`
シングルスペース レジスタ（表 A8-11 参照）。

`{<Dd[x]>, <Dd+2[x]>, <Dd+4[x]>}`
ダブルスペース レジスタ（表 A8-11 参照）。`<size>` == 8 の場合には使用できません。

`<Rn>` アクセス用のベースアドレス。

`!` 存在する場合、ライトバックが指定されます。

`<Rm>` アクセス後に適用されるアドレスオフセット

`<Rn>`、`!`、および `<Rm>` の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

表 A8-11 インデクスとレジスタスペーシングのエンコード

	`<size>` == 8	`<size>` == 16	`<size>` == 32
インデクス	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
シングルスペーシング	index_align[0] = 0	index_align[1:0] = ’00’	index_align[2:0] = ’000’
ダブルスペーシング	–	index_align[1:0] = ’10’	index_align[2:0] = ’100’

アライメント

標準的なアライメント規則が適用されます。詳細については、*アンアライインド データアクセス* P. A3-5 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];
    if wback then R[n] = R[n] + (if register_index then R[m] else 3*ebytes);
    MemU[address,ebytes] = Elem[D[d],index,esize];
    MemU[address+ebytes,ebytes] = Elem[D[d2],index,esize];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index,esize];
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.397 VST4 (multiple 4-element structures)

This instruction stores multiple 4-element structures to memory from four registers, with interleaving. For more information, see *Element and structure load/store instructions* on page A4-27. Every element of each register is saved. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VST4<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST4<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				type				size		align		Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				type				size		align		Rm			

```
if size == '11' then UNDEFINED;
case type of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  esize = 8 ÷ ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d4 > 31 then UNPREDICTABLE;
```

Related encodings See *Advanced SIMD element or structure load/store instructions* on page A7-27

Assembler syntax

`VST4<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VST4<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VST4<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VST4` instruction must be unconditional.

`<size>` The data size. It must be one of:

- 8 encoded as size = 0b00
- 16 encoded as size = 0b01
- 32 encoded as size = 0b10.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.397 VST4（複数の 4 エレメント構造のストア）

VST4（複数の 4 エレメント構造のストア）は、複数の 4 エレメント構造を 4 つのレジスタからメモリに、インターリーブ付きでストアします。詳細については、*エレメントおよび構造のロード／ストア命令* P. A4-27 を参照して下さい。各レジスタのすべてのエレメントが保存されます。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VST4<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST4<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				type				size		align		Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	0	0	Rn				Vd				type				size		align		Rm			

```
if size == '11' then UNDEFINED;
case type of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  esize = 8 * ebytes;  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d4 > 31 then UNPREDICTABLE;
```

関連エンコード *アドバンスト SIMD のエレメントまたは構造ロード／ストア命令* P. A7-27 を参照して下さい。

アセンブラ構文

`VST4<c><q>.<size> <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VST4<c><q>.<size> <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VST4<c><q>.<size> <list>, [<Rn>{@<align>}], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VST4` は常に無条件命令です。

`<size>` データサイズ。次のいずれかを使用できます。

- 8 size = 0b00 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`<list>` The list of registers to store. It must be one of:

`{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>}`
encoded as D:Vd = `<Dd>`, type = 0b0000

`{<Dd>, <Dd+2>, <Dd+4>, <Dd+6>}`
encoded as D:Vd = `<Dd>`, type = 0b0001.

`<Rn>` Contains the base address for the access.

`<align>` The alignment. It can be one of:

- `64` 8-byte alignment, encoded as align = 0b01.
- `128` 16-byte alignment, encoded as align = 0b10.
- `256` 32-byte alignment, encoded as align = 0b11.
- **omitted** Standard alignment, see *Unaligned data access* on page A3-5. Encoded as align = 0b00.

`!` If present, specifies writeback.

`<Rm>` Contains an address offset applied after the access.

For more information about `<Rn>`, `!`, and `<Rm>`, see *Advanced SIMD addressing mode* on page A7-30.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 32);
    for e = 0 to elements-1
        MemU[address,ebytes] = Elem[D[d],e,esize];
        MemU[address+ebytes,ebytes] = Elem[D[d2],e,esize];
        MemU[address+2*ebytes,ebytes] = Elem[D[d3],e,esize];
        MemU[address+3*ebytes,ebytes] = Elem[D[d4],e,esize];
        address = address + 4*ebytes;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

`<list>` ストアするレジスタのリスト。次のいずれかを使用できます。

`{<Dd>, <Dd+1>, <Dd+2>, <Dd+3>}`
D:Vd = `<Dd>`、type = 0b0000 としてエンコードされます。

`{<Dd>, <Dd+2>, <Dd+4>, <Dd+6>}`
D:Vd = `<Dd>`、type = 0b0001 としてエンコードされます。

`<Rn>` アクセス用のベースアドレス。

`<align>` アライメント。次のいずれかを使用できます。

- `64` 8 バイトアライメント。align = 0b01 としてエンコードされます。
- `128` 16 バイトアライメント。align = 0b10 としてエンコードされます。
- `256` 32 バイトアライメント。align = 0b11 としてエンコードされます。
- **省略** 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。align = 0b00 としてエンコードされます。

`!` 存在する場合、ライトバックが指定されます。

`<Rm>` アクセス後に適用されるアドレスオフセット

`<Rn>`、`!`、および `<Rm>` の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 32);
    for e = 0 to elements-1
        MemU[address,ebytes] = Elem[D[d],e,esize];
        MemU[address+ebytes,ebytes] = Elem[D[d2],e,esize];
        MemU[address+2*ebytes,ebytes] = Elem[D[d3],e,esize];
        MemU[address+3*ebytes,ebytes] = Elem[D[d4],e,esize];
        address = address + 4*ebytes;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.398 VST4 (single 4-element structure from one lane)

This instruction stores one 4-element structure to memory from corresponding elements of four registers. For details of the addressing mode see *Advanced SIMD addressing mode* on page A7-30.

Encoding T1 / A1 Advanced SIMD

```
VST4<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST4<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				size		1	1	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				size		1	1	index_align				Rm			

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '01'
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
    when '10'
        if index_align<1:0> == '11' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d4 > 31 then UNPREDICTABLE;
```

Assembler syntax

`VST4<c><q>.<size>  <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VST4<c><q>.<size>  <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VST4<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>`	Rm = other values

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VST4` instruction must be unconditional.

`<size>` The data size. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.398 VST4（1 つのレーンからの単一の 4 エレメント構造のストア）

VST4（1 つのレーンからの単一の 4 エレメント構造のストア）は、1 つの 4 エレメント構造を、4 つのレジスタの対応するエレメントからメモリにストアします。アドレシングモードの詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

エンコード T1/A1 アドバンスト SIMD

```
VST4<c>.<size> <list>, [<Rn>{@<align>}]{!}
VST4<c>.<size> <list>, [<Rn>{@<align>}], <Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	0	0	Rn				Vd				size		1	1	index_align				Rm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0	Rn				Vd				size		1	1	index_align				Rm			

```
if size == '11' then UNDEFINED;
case size of
    when '00'
        ebytes = 1;  esize = 8;  index = UInt(index_align<3:1>);  inc = 1;
        alignment = if index_align<0> == '0' then 1 else 4;
    when '01'
        ebytes = 2;  esize = 16;  index = UInt(index_align<3:2>);
        inc = if index_align<1> == '0' then 1 else 2;
        alignment = if index_align<0> == '0' then 1 else 8;
    when '10'
        if index_align<1:0> == '11' then UNDEFINED;
        ebytes = 4;  esize = 32;  index = UInt(index_align<3>);
        inc = if index_align<2> == '0' then 1 else 2;
        alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if d4 > 31 then UNPREDICTABLE;
```

アセンブラ構文

`VST4<c><q>.<size>  <list>, [<Rn>{@<align>}]`	Rm = '1111'
`VST4<c><q>.<size>  <list>, [<Rn>{@<align>}]!`	Rm = '1101'
`VST4<c><q>.<size>  <list>, [<Rn>{@<align>}], <Rm>`	Rm = その他の値

各項目の説明については以下を参照して下さい。

`<c><q>.` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VST4` は常に無条件命令です。

`<size>.` データサイズ。次のいずれかを使用できます。

- `8 .` size = 0b00 としてエンコードされます。
- `16 .` size = 0b01 としてエンコードされます。
- `32 .` size = 0b10 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<list> The registers containing the structure. Encoded with D:Vd = <Dd>. It must be one of:

{<Dd[x]>, <Dd+1[x]>, <Dd+2[x]>, <Dd+3[x]>}
Single-spaced registers, see Table A8-12.

{<Dd[x]>, <Dd+2[x]>, <Dd+4[x]>, <Dd+6[x]>}
Double-spaced registers, see Table A8-12. This is not available if <size> == 8.

<Rn> The base address for the access.

<align> The alignment. It can be:

- 32 4-byte alignment, available only if <size> is 8.
- 64 8-byte alignment, available only if <size> is 16 or 32.
- 128 16-byte alignment, available only if <size> is 32.
- **omitted** Standard alignment, see *Unaligned data access* on page A3-5.

! If present, specifies writeback.

<Rm> Contains an address offset applied after the access.

For more information about <Rn>, !, and <Rm>, see *Advanced SIMD addressing mode* on page A7-30.

Table A8-12 Encoding of index, alignment, and register spacing

	<size> == 8	<size> == 16	<size> == 32
Index	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
Single-spacing	-	index_align[1] = 0	index_align[2] = 0
Double-spacing	-	index_align[1] = 1	index_align[2] = 1
<align> omitted	index_align[0] = 0	index_align[0] = 0	index_align[1:0] = '00'
<align> == 32	index_align[0] = 1	-	-
<align> == 64	-	index_align[0] = 1	index_align[1:0] = '01'
<align> == 128	-	-	index_align[1:0] = '10'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 4*ebytes);
    MemU[address,ebytes] = Elem[D[d],index,esize];
    MemU[address+ebytes,ebytes] = Elem[D[d2],index,esize];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index,esize];
    MemU[address+3*ebytes,ebytes] = Elem[D[d4],index,esize];
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<list>. 構造を含むレジスタ。D:Vd = <Dd> としてエンコードされます。次のいずれかを使用できます。

{<Dd[x]>, <Dd+1[x]>, <Dd+2[x]>, <Dd+3[x]>}
シングルスペース レジスタ（表 A8-12 参照）。

{<Dd[x]>, <Dd+2[x]>, <Dd+4[x]>, <Dd+6[x]>}
ダブルスペース レジスタ（表 A8-12 参照）。<size> == 8 の場合には使用できません。

<Rn>. アクセス用のベースアドレス。

<align>. アライメント。次のいずれかを使用できます。

- 32 . 4 バイトアライメント。<size> が 8 の場合にのみ使用できます。
- 64 . 8 バイトアライメント。<size> が 16 または 32 の場合にのみ使用できます。
- 128 . 16 バイトアライメント。<size> が 32 の場合にのみ使用できます。
- **省略** . 標準的なアライメント。詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。

!. 存在する場合、ライトバックが指定されます。

<Rm>. アクセス後に適用されるアドレスオフセット

<Rn>、!、および <Rm> の詳細については、*アドバンスト SIMD のアドレシングモード* P. A7-30 を参照して下さい。

表 A8-12 インデクス、アライメント、およびレジスタスペーシングのエンコード

	<size> == 8	<size> == 16	<size> == 32
インデクス	index_align[3:1] = x	index_align[3:2] = x	index_align[3] = x
シングルスペーシング	–	index_align[1] = 0	index_align[2] = 0
ダブルスペーシング	–	index_align[1] = 1	index_align[2] = 1
<align> 省略	index_align[0] = 0	index_align[0] = 0	index_align[1:0] = '00'
<align> == 32	index_align[0] = 1	–	–
<align> == 64	–	index_align[0] = 1	index_align[1:0] = '01'
<align> == 128	–	–	index_align[1:0] = '10'

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();  NullCheckIfThumbEE(n);
    address = R[n];  if (address MOD alignment) != 0 then GenerateAlignmentException();
    if wback then R[n] = R[n] + (if register_index then R[m] else 4*ebytes);
    MemU[address,ebytes] = Elem[D[d],index,esize];
    MemU[address+ebytes,ebytes] = Elem[D[d2],index,esize];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index,esize];
    MemU[address+3*ebytes,ebytes] = Elem[D[d4],index,esize];
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.399 VSTM

Vector Store Multiple stores multiple extension registers to consecutive memory locations using an address from an ARM core register.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD

`VSTM{mode}<c> <Rn>{!}, <list>` `<list>` is consecutive 64-bit registers

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0	1 1 0	P	U	D	W	0	Rn	Vd	1 0 1 1	imm8

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
cond	1 1 0	P	U	D	W	0	Rn	Vd	1 0 1 1	imm8

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && U == '0' && W == '1' && Rn == '1101' then SEE VPUSH;
if P == '1' && W == '0' then SEE VSTR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FSTMX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
```

Encoding T2 / A2 VFPv2, VFPv3

`VSTM{mode}<c> <Rn>{!}, <list>` `<list>` is consecutive 32-bit registers

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0	1 1 0	P	U	D	W	0	Rn	Vd	1 0 1 0	imm8

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
cond	1 1 0	P	U	D	W	0	Rn	Vd	1 0 1 0	imm8

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && U == '0' && W == '1' && Rn == '1101' then SEE VPUSH;
if P == '1' && W == '0' then SEE VSTR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

Related encodings See *64-bit transfers between ARM core and extension registers* on page A7-32

FSTMX Encoding T1/A1 behaves as described by the pseudocode if imm8 is odd. However, there is no UAL syntax for such encodings and their use is deprecated. For more information, see *FLDMX, FSTMX* on page A8-101.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.399 VSTM

VSTM（ベクタ複数ストア）は、ARM コアレジスタからのアドレスを使用して、複数の拡張機能レジスタを、連続するメモリ位置にストアします。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD

`VSTM{mode}<c> <Rn>{!}, <list>` `<list>` は連続する 64 ビットレジスタ

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0	1 1 0	P	U	D	W	0	Rn	Vd	1 0 1 1	imm8

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
cond	1 1 0	P	U	D	W	0	Rn	Vd	1 0 1 1	imm8

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && U == '0' && W == '1' && Rn == '1101' then SEE VPUSH;
if P == '1' && W == '0' then SEE VSTR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;  // If UInt(imm8) is odd, see "FSTMX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
```

エンコード T2/A2 VFPv2、VFPv3

`VSTM{mode}<c> <Rn>{!}, <list>` `<list>` は連続する 32 ビットレジスタ

15 14 13 12	11 10 9	8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0	1 1 0	P	U	D	W	0	Rn	Vd	1 0 1 0	imm8

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
cond	1 1 0	P	U	D	W	0	Rn	Vd	1 0 1 0	imm8

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && U == '0' && W == '1' && Rn == '1101' then SEE VPUSH;
if P == '1' && W == '0' then SEE VSTR;
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

関連エンコード *ARM コアレジスタと拡張機能レジスタとの 64 ビット転送* P. A7-32 を参照して下さい。

FSTMX imm8 が奇数の場合、エンコード T1/A1 は擬似コードで記述されているように動作します。しかし、このようなエンコード用の UAL 構文は存在せず、このようなエンコードの使用は推奨されません。詳細については、*FLDMX*、*FSTMX* P. A8-101 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VSTM{<mode>}<c><q>{.<size>}  <Rn>{!}, <list>
```

where:

`<mode>` The addressing mode:

- `IA` Increment After. The consecutive addresses start at the address specified in `<Rn>`. This is the default and can be omitted. Encoded as P = 0, U = 1.
- `DB` Decrement Before. The consecutive addresses end just before the address specified in `<Rn>`. Encoded as P = 1, U = 0.

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<size>` An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers in `<list>`.

`<Rn>` The base register. The SP can be used. In the ARM instruction set, if ! is not specified the PC can be used. However, use of the PC is deprecated.

`!` Causes the instruction to write a modified value back to `<Rn>`. Required if <mode> == `DB`. Encoded as W = 1.

If ! is omitted, the instruction does not change `<Rn>` in this way. Encoded as W = 0.

`<list>` The extension registers to be stored, as a list of consecutively numbered doubleword (encoding T1 / A1) or singleword (encoding T2 / A2) registers, separated by commas and surrounded by brackets. It is encoded in the instruction by setting D and Vd to specify the first register in the list, and imm8 to twice the number of registers in the list (encoding T1 / A1) or the number of registers (encoding T2 / A2). `<list>` must contain at least one register. If it contains doubleword registers it must not contain more than 16 registers.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);  NullCheckIfThumbEE(n);
    address = if add then R[n] else R[n]-imm32;
    if wback then R[n] = if add then R[n}+imm32 else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = S[d+r];  address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian() then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian() then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VSTM{<mode>}<c><q>{.<size>}  <Rn>{!}, <list>
```

各項目の説明については以下を参照して下さい。

`<mode>` アドレシングモード。

- `IA` ポストインクリメント。連続するアドレスは、`<Rn>` で指定されたアドレスから始まります。これはデフォルトであり、省略することができます。P = 0、U = 1 としてエンコードされます。
- `DB` プリデクリメント。連続するアドレスは、`<Rn>` で指定されたアドレスの 1 つ前で終わります。P = 1、U = 0 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<size>` オプションのデータサイズ指定子。指定する場合、`<list>` に含まれるレジスタのサイズ（32 ビットまたは 64 ビット）に等しいことが必要です。

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。ARM 命令セットでは、! が指定されていない場合に PC を使用できます。しかし、PC の使用は推奨されません。

`!` 命令の実行によって変更された値を `<Rn>` に書き戻すように指示します。〈mode〉 == `DB` の場合に必須です。W = 1 としてエンコードされます。

! を省略した場合は、命令によって `<Rn>` は変更されません。W = 0 としてエンコードされます。

`<list>` ストアされる拡張機能レジスタ。カンマで区切られ、角括弧で囲まれた、番号が連続しているダブルワード（エンコード T1/A1）レジスタまたはシングルワード（エンコード T2/A2）レジスタのリストとして示されます。リストの最初のレジスタを指定するために D および Vd を設定し、リストに含まれるレジスタの数の 2 倍（エンコード T1/A1）またはリストに含まれるレジスタの数（エンコード T2/A2）を指定するために imm8 を設定することで、命令にエンコードされます。`<list>` には、少なくとも 1 つのレジスタを含む必要があります。ダブルワードレジスタが含まれる場合、16 を超えるレジスタを含むことはできません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);  NullCheckIfThumbEE(n);
    address = if add then R[n] else R[n]-imm32;
    if wback then R[n] = if add then R[n}+imm32 else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = S[d+r];  address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian() then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian() then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.400 VSTR

This instruction stores a single extension register to memory, using an address from an ARM core register, with an optional offset.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD

```
VSTR<c> <Dd>, [<Rn>{, #+/-<imm>}]
```

15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0	1 1 0 1	U	D	0	0	Rn	Vd	1 0 1 1	imm8

31 30 29 28	27 26 25 24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
cond	1 1 0 1	U	D	0	0	Rn	Vd	1 0 1 1	imm8

```
single_reg = FALSE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(D:Vd);  n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_ARM then UNPREDICTABLE;
```

Encoding T2 / A2 VFPv2, VFPv3

```
VSTR<c> <Sd>, [<Rn>{, #+/-<imm>}]
```

15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0	1 1 0 1	U	D	0	0	Rn	Vd	1 0 1 0	imm8

31 30 29 28	27 26 25 24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
cond	1 1 0 1	U	D	0	0	Rn	Vd	1 0 1 0	imm8

```
single_reg = TRUE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(Vd:D);  n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_ARM then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.400 VSTR

VSTR（拡張レジスタのストア）は、ARM コアレジスタからのアドレスとオプションのオフセットを使用して、1 つの拡張機能レジスタをメモリにストアします。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD

```
VSTR<c> <Dd>, [<Rn>{, #+/-<imm>}]
```

15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0	1 1 0 1	U	D	0	0	Rn	Vd	1 0 1 1	imm8

31 30 29 28	27 26 25 24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
cond	1 1 0 1	U	D	0	0	Rn	Vd	1 0 1 1	imm8

```
single_reg = FALSE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(D:Vd);  n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_ARM then UNPREDICTABLE;
```

エンコード T2/A2 VFPv2、VFPv3

```
VSTR<c> <Sd>, [<Rn>{, #+/-<imm>}]
```

15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0	1 1 0 1	U	D	0	0	Rn	Vd	1 0 1 0	imm8

31 30 29 28	27 26 25 24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
cond	1 1 0 1	U	D	0	0	Rn	Vd	1 0 1 0	imm8

```
single_reg = TRUE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(Vd:D);  n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_ARM then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VSTR<c><q>{.64}  <Dd>, [<Rn>{, #+/-<imm>}]          Encoding T1 / A1
VSTR<c><q>{.32}  <Sd>, [<Rn>{, #+/-<imm>}]          Encoding T2 / A2
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`.32, .64` Optional data size specifiers.

`<Dd>` The source register for a doubleword store.

`<Sd>` The source register for a singleword store.

`<Rn>` The base register. The SP can be used. In the ARM instruction set the PC can be used. However, use of the PC is deprecated.

`+/-` Is + or omitted if the immediate offset is to be added to the base register value (`add == TRUE`), or – if it is to be subtracted (`add == FALSE`). `#0` and `#-0` generate different instructions.

`<imm>` The immediate offset used to form the address. Values are multiples of 4 in the range 0-1020. `<imm>` can be omitted, meaning an offset of +0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);  NullCheckIfThumbEE(n);
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    if single_reg then
        MemA[address,4] = S[d];
    else
        // Store as two word-aligned words in the correct order for current endianness.
        MemA[address,4] = if BigEndian() then D[d]<63:32> else D[d]<31:0>;
        MemA[address+4,4] = if BigEndian() then D[d]<31:0> else D[d]<63:32>;
```

Exceptions

Undefined Instruction, Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
VSTR<c><q>{.64}  <Dd>, [<Rn>{, #+/-<imm>}]          エンコード T1/A1
VSTR<c><q>{.32}  <Sd>, [<Rn>{, #+/-<imm>}]          エンコード T2/A2
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`.32, .64` オプションのデータサイズ指定子。

`<Dd>` ダブルワードストアの場合のソースレジスタ

`<Sd>` シングルワードストアの場合のソースレジスタ

`<Rn>` ベースレジスタ。このレジスタには SP を指定できます。ARM 命令セットでは、PC を指定できます。しかし、PC の使用は推奨されません。

`+/-` イミディエートオフセットをベースレジスタの値に加算する場合 (`add == TRUE`) は、「+」にするか、省略します。減算する場合 (`add == FALSE`) は、「-」にします。`#0` と `#-0` では異なる命令が生成されます。

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。値は、0 ～ 1020 までの 4 の倍数です。`<imm>` は省略可能で、その場合オフセットは 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);  NullCheckIfThumbEE(n);
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    if single_reg then
        MemA[address,4] = S[d];
    else
        // Store as two word-aligned words in the correct order for current endianness.
        MemA[address,4] = if BigEndian() then D[d]<63:32> else D[d]<31:0>;
        MemA[address+4,4] = if BigEndian() then D[d]<31:0> else D[d]<63:32>;
```

例外

未定義命令、データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.401 VSUB (integer)

Vector Subtract subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Encoding T1 / A1 Advanced SIMD

```
VSUB<c>.<dt> <Qd>, <Qn>, <Qm>
VSUB<c>.<dt> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.401 VSUB（整数）

VSUB（ベクタ減算（整数））は、一方のベクタのエレメントを、他方のベクタの対応するエレメントから減算し、結果をデスティネーションベクタに配置します。

エンコード T1/A1 アドバンスト SIMD

```
VSUB<c>.<dt> <Qd>, <Qn>, <Qm>
VSUB<c>.<dt> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VSUB<c><q>.<dt>  {<Qd>,} <Qn>, <Qm>
VSUB<c><q>.<dt>  {<Dd>,} <Dn>, <Dm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VSUB` instruction must be unconditional.
`<dt>`	The data type for the elements of the vectors. It must be one of: `I8` size = 0b00 `I16` size = 0b01 `I32` size = 0b10 `I64` size = 0b11.
`<Qd>, <Qn>, <Qm>`	The destination vector and the operand vectors, for a quadword operation.
`<Dd>, <Dn>, <Dm>`	The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] - Elem[D[m+r],e,esize];
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VSUB<c><q>.<dt>  {<Qd>,} <Qn>, <Qm>
VSUB<c><q>.<dt>  {<Dd>,} <Dn>, <Dm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VSUB` は常に無条件命令です。
`<dt>`	ベクタのエレメントのデータ型。次のいずれかを使用できます。 `I8` size = 0b00 `I16` size = 0b01 `I32` size = 0b10 `I64` size = 0b11
`<Qd>, <Qn>, <Qm>`	クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Dd>, <Dn>, <Dm>`	ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] - Elem[D[m+r],e,esize];
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.402 VSUB (floating-point)

Vector Subtract subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Encoding T1 / A1 Advanced SIMD (UNDEFINED in integer-only variant)

```
VSUB<c>.F32 <Qd>, <Qn>, <Qm>
VSUB<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Encoding T2 / A2 VFPv2, VFPv3 (sz = 1 UNDEFINED in single-precision only variants)

```
VSUB<c>.F64 <Dd>, <Dn>, <Dm>
VSUB<c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	1	1	Vn				Vd				1	0	1	sz	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	D	1	1	Vn				Vd				1	0	1	sz	N	1	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP vectors Encoding T2 / A2 can operate on VFP vectors under control of the FPSCR.LEN and FPSCR.STRIDE bits. For details see Appendix F *VFP Vector Operation Support*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.402 VSUB（浮動小数点数）

VSUB（ベクタ減算（浮動小数点数））は、一方のベクタのエレメントを、他方のベクタの対応するエレメントから減算し、結果をデスティネーションベクタに配置します。

エンコード T1/A1 アドバンスト SIMD（整数のみのバリアントでは未定義）

```
VSUB<c>.F32 <Qd>, <Qn>, <Qm>
VSUB<c>.F32 <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' then UNDEFINED;
advsimd = TRUE;  esize = 32;  elements = 2;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

エンコード T2/A2 VFPv2、VFPv3（単精度のみのバリアントでは sz = 1 は未定義）

```
VSUB<c>.F64 <Dd>, <Dn>, <Dm>
VSUB<c>.F32 <Sd>, <Sn>, <Sm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	1	1	Vn				Vd				1	0	1	sz	N	1	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	0	D	1	1	Vn				Vd				1	0	1	sz	N	1	M	0	Vm			

```
if FPSCR.LEN != '000' || FPSCR.STRIDE != '00' then SEE "VFP vectors";
advsimd = FALSE;  dp_operation = (sz == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

VFP ベクタ エンコード T2/A2 は、FPSCR.LEN ビットと FPSCR.STRIDE ビットの制御の下で VFP ベクタに作用することができます。詳細については、付録 F *「VFP ベクタ演算のサポート」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VSUB<c><q>.F32 {<Qd>,} <Qn>, <Qm>`	Encoding T1 / A1, Q = 1, sz = 0
`VSUB<c><q>.F32 {<Dd>,} <Dn>, <Dm>`	Encoding T1 / A1, Q = 0, sz = 0
`VSUB<c><q>.F64 {<Dd>,} <Dn>, <Dm>`	Encoding T2 / A2, sz = 1
`VSUB<c><q>.F32 {<Sd>,} <Sn>, <Sm>`	Encoding T2 / A2, sz = 0

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM Advanced SIMD `VSUB` instruction must be unconditional.
`<Qd>, <Qn>, <Qm>`	The destination vector and the operand vectors, for a quadword operation.
`<Dd>, <Dn>, <Dm>`	The destination vector and the operand vectors, for a doubleword operation.
`<Sd>, <Sn>, <Sm>`	The destination vector and the operand vectors, for a singleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPSub(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], FALSE);
    else            // VFP instruction
        if dp_operation then
            D[d] = FPSub(D[n], D[m], TRUE);
        else
            S[d] = FPSub(S[n], S[m], TRUE);
```

Exceptions

Undefined Instruction.

Floating-point exceptions: Input Denormal, Invalid Operation, Overflow, Underflow, and Inexact.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VSUB<c><q>.F32 {<Qd>,} <Qn>, <Qm>`	エンコード T1/A1、Q = 1、sz = 0
`VSUB<c><q>.F32 {<Dd>,} <Dn>, <Dm>`	エンコード T1/A1、Q = 0、sz = 0
`VSUB<c><q>.F64 {<Dd>,} <Dn>, <Dm>`	エンコード T2/A2、sz = 1
`VSUB<c><q>.F32 {<Sd>,} <Sn>, <Sm>`	エンコード T2/A2、sz = 0

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM アドバンスト SIMD `VSUB` は常に無条件命令です。
`<Qd>, <Qn>, <Qm>`	クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Dd>, <Dn>, <Dm>`	ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ
`<Sd>, <Sn>, <Sm>`	シングルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPSub(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], FALSE);
    else            // VFP instruction
        if dp_operation then
            D[d] = FPSub(D[n], D[m], TRUE);
        else
            S[d] = FPSub(S[n], S[m], TRUE);
```

例外

未定義命令

浮動小数点例外：入力非正規化、無効演算、オーバフロー、アンダーフロー、不正確

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.403 VSUBHN

Vector Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector takes the most significant half of each result, and places the final results in a doubleword vector. The results are truncated. (For rounded results, see *VRSUBHN* on page A8-748.

There is no distinction between signed and unsigned integers.

Encoding T1 / A1 Advanced SIMD

```
VSUBHN<c>.<dt> <Dd>, <Qn>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	size		Vn				Vd				0	1	1	0	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	size		Vn				Vd				0	1	1	0	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.403 VSUBHN

VSUBHN（上位半分を返すベクタ減算および縮小）は、一方のクワッドワードベクタのエレメントを、他方のクワッドワードベクタの対応するエレメントから減算し、各結果の上位半分を取得し、最終結果をダブルワードベクタに配置します。結果は切り捨てられます（丸められる結果については、*VRSUBHN* P. A8-748 を参照して下さい）。

整数には符号付きと符号なしの区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VSUBHN<c>.<dt> <Dd>, <Qn>, <Qm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	size		Vn				Vd				0	1	1	0	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	size		Vn				Vd				0	1	1	0	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VSUBHN<c><q>.<dt>  <Dd>, <Qn>, <Qm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7. An ARM `VSUBHN` instruction must be unconditional.
`<dt>`	The data type for the elements of the operands. It must be one of: `I16` size = 0b00 `I32` size = 0b01 `I64` size = 0b10.
`<Dd>, <Qn>, <Qm>`	The destination vector, the first operand vector, and the second operand vector.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Elem[Q[n>>1],e,2*esize] - Elem[Q[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
VSUBHN<c><q>.<dt>  <Dd>, <Qn>, <Qm>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VSUBHN` は常に無条件命令です。
`<dt>`	オペランドのエレメントのデータ型。次のいずれかを使用できます。 `I16` size = 0b00 `I32` size = 0b01 `I64` size = 0b10
`<Dd>, <Qn>, <Qm>`	デスティネーションベクタ、最初のオペランドベクタ、および 2 番目のオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Elem[Q[n>>1],e,2*esize] - Elem[Q[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.404 VSUBL, VSUBW

Vector Subtract Long subtracts the elements of one doubleword vector from the corresponding elements of another doubleword vector, and places the results in a quadword vector. Before subtracting, it sign-extends or zero-extends the elements of both operands.

Vector Subtract Wide subtracts the elements of a doubleword vector from the corresponding elements of a quadword vector, and places the results in another quadword vector. Before subtracting, it sign-extends or zero-extends the elements of the doubleword operand.

Encoding T1 / A1 Advanced SIMD

```
VSUBL<c>.<dt> <Qd>, <Dn>, <Dm>
VSUBW<c>.<dt> {<Qd>,} <Qn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				0	0	1	op	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				0	0	1	op	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw == (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings See *Advanced SIMD data-processing instructions* on page A7-10

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.404 VSUBL、VSUBW

VSUBL（ベクタロング減算）は、一方のダブルワードベクタのエレメントを、他方のダブルワードベクタの対応するエレメントから減算し、結果をクワッドワードベクタに配置します。減算する前に、両方のオペランドのエレメントを符号拡張またはゼロ拡張します。

VSUBW（ベクタワイド減算）は、ダブルワードベクタのエレメントを、クワッドワードベクタの対応するエレメントから減算し、結果を別のクワッドワードベクタに配置します。減算する前に、ダブルワードオペランドのエレメントを符号拡張またはゼロ拡張します。

エンコード T1/A1 アドバンスト SIMD

```
VSUBL<c>.<dt> <Qd>, <Dn>, <Dm>
VSUBW<c>.<dt> {<Qd>,} <Qn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	size		Vn				Vd				0	0	1	op	N	0	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	size		Vn				Vd				0	0	1	op	N	0	M	0	Vm			

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw == (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

関連エンコード *アドバンスト SIMD のデータ処理命令* P. A7-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VSUBL<c><q>.<dt> <Qd>, <Dn>, <Dm>`	Encoded as op = 0
`VSUBW<c><q>.<dt> {<Qd>,} <Qn>, <Dm>`	Encoded as op = 1

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VSUBL` or `VSUBW` instruction must be unconditional.

`<dt>` The data type for the elements of the second operand. It must be one of:

- `S8` encoded as size = 0b00, U = 0
- `S16` encoded as size = 0b01, U = 0
- `S32` encoded as size = 0b10, U = 0
- `U8` encoded as size = 0b00, U = 1
- `U16` encoded as size = 0b01, U = 1
- `U32` encoded as size = 0b10, U = 1.

`<Qd>` The destination register.

`<Qn>, <Dm>` The first and second operand registers for a `VSUBW` instruction.

`<Dn>, <Dm>` The first and second operand registers for a `VSUBL` instruction.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vsubw then
            op1 = Int(Elem[Q[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[D[n],e,esize], unsigned);
        result = op1 - Int(Elem[D[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VSUBL<c><q>.<dt> <Qd>, <Dn>, <Dm>`	op = 0 としてエンコードされる
`VSUBW<c><q>.<dt> {<Qd>,} <Qn>, <Dm>`	op = 1 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VSUBL` および `VSUBW` は常に無条件命令です。

`<dt>` 2 番目のオペランドのエレメントのデータ型。次のいずれかを使用できます。

- `S8` size = 0b00、U = 0 としてエンコードされます。
- `S16` size = 0b01、U = 0 としてエンコードされます。
- `S32` size = 0b10、U = 0 としてエンコードされます。
- `U8` size = 0b00、U = 1 としてエンコードされます。
- `U16` size = 0b01、U = 1 としてエンコードされます。
- `U32` size = 0b10、U = 1 としてエンコードされます。

`<Qd>` デスティネーションレジスタ

`<Qn>, <Dm>` `VSUBW` 命令の最初および 2 番目のオペランドレジスタ

`<Dn>, <Dm>` `VSUBL` 命令の最初および 2 番目のオペランドレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vsubw then
            op1 = Int(Elem[Q[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[D[n],e,esize], unsigned);
        result = op1 - Int(Elem[D[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.405 VSWP

VSWP (Vector Swap) exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.

Encoding T1 / A1 Advanced SIMD

```
VSWP<c> <Qd>, <Qm>
VSWP<c> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	0	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	0	0	0	Q	M	0	Vm			

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.405 VSWP

VSWP（ベクタスワップ）は、2 つのベクタの内容を交換します。ベクタには、ダブルワードまたはクワッドワードを使用できます。これらにはデータ型の区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VSWP<c> <Qd>, <Qm>
VSWP<c> <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	0	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	0	0	0	Q	M	0	Vm			

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VSWP<c><q>{.<dt>}  <Qd>, <Qm>`	Encoded as Q = 1, size = '00'
`VSWP<c><q>{.<dt>}  <Dd>, <Dm>`	Encoded as Q = 0, size = '00'

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VSWP` instruction must be unconditional.

`<dt>` An optional data type. It is ignored by assemblers, and does not affect the encoding.

`<Qd>, <Qm>` The vectors for a quadword operation.

`<Dd>, <Dm>` The vectors for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            tmp = D[d+r];
            D[d+r] = D[m+r];
            D[m+r] = tmp;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VSWP<c><q>{.<dt>}  <Qd>, <Qm>`	Q = 1、size = '00' としてエンコードされる
`VSWP<c><q>{.<dt>}  <Dd>, <Dm>`	Q = 0、size = '00' としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VSWP` は常に無条件命令です。

`<dt>` オプションのデータ型。アセンブラによって無視され、エンコードに影響を与えません。

`<Qd>, <Qm>` クワッドワード演算の場合のベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            tmp = D[d+r];
            D[d+r] = D[m+r];
            D[m+r] = tmp;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.406 VTBL, VTBX

Vector Table Lookup uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0.

Vector Table Extension works in the same way, except that indexes out of range leave the destination element unchanged.

Encoding T1 / A1 Advanced SIMD

```
V<op><c>.8 <Dd>, <list>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	Vn				Vd				1	0	len		N	op	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	Vn				Vd				1	0	len		N	op	M	0	Vm			

```
is_vtbl = (op == '0');  length = UInt(len)+1;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.406 VTBL、VTBX

VTBL（ベクタ テーブルルックアップ）は、制御ベクタのバイトインデクスを使用して、テーブル内のバイト値をルックアップし、新しいベクタを生成します。インデクスが範囲外の場合、0 が返されます。

VTBX（ベクタテーブル拡張）は、VTBL と同様に機能しますが、インデクスが範囲外の場合、デスティネーションエレメントは変更されません。

エンコード T1/A1 アドバンスト SIMD

```
V<op><c>.8 <Dd>, <list>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	Vn				Vd				1	0	len		N	op	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	Vn				Vd				1	0	len		N	op	M	0	Vm			

```
is_vtbl = (op == '0');  length = UInt(len)+1;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
V<op><c><q>.8  <Dd>, <list>, <Dm>
```

where:

`<op>` Specifies the operation. It must be one of:

- `TBL` encoded as op = 0
- `TBX` encoded as op = 1

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VTBL` or `VTBX` instruction must be unconditional.

`<Dd>` The destination vector.

`<list>` The vectors containing the table. It must be one of:

- `{<Dn>}` encoded as len = 0b00
- `{<Dn>,<Dn+1>}` encoded as len = 0b01
- `{<Dn>,<Dn+1>,<Dn+2>}` encoded as len = 0b10
- `{<Dn>,<Dn+1>,<Dn+2>,<Dn+3>}` encoded as len = 0b11

`<Dm>` The index vector.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    // Create 256-bit = 32-byte table variable, with zeros in entries that will not be used.
    table3 = if length == 4 then D[n+3] else Zeros(64);
    table2 = if length >= 3 then D[n+2] else Zeros(64);
    table1 = if length >= 2 then D[n+1] else Zeros(64);
    table = table3 : table2 : table1 : D[n];

    for i = 0 to 7
        index = UInt(Elem[D[m],i,8]);
        if index < 8*length then
            Elem[D[d],i,8] = Elem[table,index,8];
        else
            if is_vtbl then
                Elem[D[d],i,8] = Zeros(8);
            // else Elem[D[d],i,8] unchanged
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
V<op><c><q>.8  <Dd>, <list>, <Dm>
```

各項目の説明については以下を参照して下さい。

`<op>` 演算。次のいずれかを使用できます。

- `TBL` op = 0 としてエンコードされます。
- `TBX` op = 1 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。ARM `VTBL` および `VTBX` は常に無条件命令です。

`<Dd>` デスティネーションベクタ

`<list>` テーブルを含むベクタ。次のいずれかを使用できます。

- `{<Dn>}` len = 0b00 としてエンコードされます。
- `{<Dn>,<Dn+1>}` len = 0b01 としてエンコードされます。
- `{<Dn>,<Dn+1>,<Dn+2>}` len = 0b10 としてエンコードされます。
- `{<Dn>,<Dn+1>,<Dn+2>,<Dn+3>}` len = 0b11 としてエンコードされます。

`<Dm>` インデクスベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    // Create 256-bit = 32-byte table variable, with zeros in entries that will not be used.
    table3 = if length == 4 then D[n+3] else Zeros(64);
    table2 = if length >= 3 then D[n+2] else Zeros(64);
    table1 = if length >= 2 then D[n+1] else Zeros(64);
    table = table3 : table2 : table1 : D[n];

    for i = 0 to 7
        index = UInt(Elem[D[m],i,8]);
        if index < 8*length then
            Elem[D[d],i,8] = Elem[table,index,8];
        else
            if is_vtbl then
                Elem[D[d],i,8] = Zeros(8);
            // else Elem[D[d],i,8] unchanged
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.407 VTRN

Vector Transpose treats the elements of its operand vectors as elements of 2 × 2 matrices, and transposes the matrices.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

Encoding T1 / A1 Advanced SIMD

```
VTRN<c>.<size>  <Qd>, <Qm>
VTRN<c>.<size>  <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	0	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	0	0	1	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Figure A8-7 shows the operation of doubleword VTRN. Quadword VTRN performs the same operation as doubleword VTRN twice, once on the upper halves of the quadword vectors, and once on the lower halves

Figure A8-7 Operation of doubleword VTRN

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.407 VTRN

VTRN（ベクタ転置）は、オペランドベクタのエレメントを 2 × 2 行列のエレメントとして処理し、行列を転置します。

ベクタのエレメントには、8 ビット、16 ビット、または 32 ビットを使用できます。これらにはデータ型の区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VTRN<c>.<size>  <Qd>, <Qm>
VTRN<c>.<size>  <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	0	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	0	0	1	Q	M	0	Vm			

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ダブルワード VTRN の動作を図 A8-7 に示します。クワッドワード VTRN は、ダブルワード VTRN を 2 回（クワッドワードベクタの上位半分と下位半分に対して）実行するのと同じ動作を実行します。

図 A8-7 ダブルワード VTRN の動作

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VTRN<c><q>.<size>  <Qd>, <Qm>`	Encoded as Q = 1
`VTRN<c><q>.<size>  <Dd>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VTRN` instruction must be unconditional.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

`<Qd>, <Qm>` The destination vector, and the operand vector, for a quadword operation.

`<Dd>, <Dm>` The destination vector, and the operand vector, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements/2;

    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            for e = 0 to h-1
                tmp = Elem[D[d+r],2*e+1,esize];
                Elem[D[d+r],2*e+1,esize] = Elem[D[m+r],2*e,esize];
                Elem[D[m+r],2*e,esize] = tmp;
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VTRN<c><q>.<size>  <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VTRN<c><q>.<size>  <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VTRN` は常に無条件命令です。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

`<Qd>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements/2;

    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            for e = 0 to h-1
                tmp = Elem[D[d+r],2*e+1,esize];
                Elem[D[d+r],2*e+1,esize] = Elem[D[m+r],2*e,esize];
                Elem[D[m+r],2*e,esize] = tmp;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.408 VTST

Vector Test Bits takes each element in a vector, and bitwise ANDs it with the corresponding element of a second vector. If the result is not zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

- 8-bit, 16-bit, or 32-bit fields.

The result vector elements are bitfields the same size as the operand vector elements.

Encoding T1 / A1 Advanced SIMD

```
VTST<c>.<size> <Qd>, <Qn>, <Qm>
VTST<c>.<size> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.408 VTST

VTST（ベクタ テストビット）は、ベクタの各エレメントを取得して、2 番目のベクタの対応するエレメントとのビット単位の論理積を実行します。結果が 0 以外の場合、デスティネーションベクタの対応するエレメントがすべて 1 にセットされます。それ以外の場合、すべて 0 にセットされます。

オペランドベクタのエレメントには、次のいずれかを使用できます。

- 8 ビット、16 ビット、または 32 ビットのフィールド

結果ベクタのエレメントはビットフィールドで、オペランドベクタのエレメントと同じサイズです。

エンコード T1/A1 アドバンスト SIMD

```
VTST<c>.<size> <Qd>, <Qn>, <Qm>
VTST<c>.<size> <Dd>, <Dn>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size		Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VTST<c><q>.<size> {<Qd>,} <Qn>, <Qm>`	Encoded as Q = 1
`VTST<c><q>.<size> {<Dd>,} <Dn>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VTST` instruction must be unconditional.

`<size>` The data size for the elements of the operands. It must be one of:

- `8` encoded as size = 0b00
- `16` encoded as size = 0b01
- `32` encoded as size = 0b10.

`<Qd>, <Qn>, <Qm>` The destination vector and the operand vectors, for a quadword operation.

`<Dd>, <Dn>, <Dm>` The destination vector and the operand vectors, for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !IsZero(Elem[D[n+r],e,esize] AND Elem[D[m+r],e,esize]) then
                Elem[D[d+r],e,esize] = Ones(esize);
            else
                Elem[D[d+r],e,esize] = Zeros(esize);
```

Exceptions

Undefined Instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VTST<c><q>.<size> {<Qd>,} <Qn>, <Qm>`	Q = 1 としてエンコードされる
`VTST<c><q>.<size> {<Dd>,} <Dn>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VTST` は常に無条件命令です。

`<size>` オペランドのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` size = 0b10 としてエンコードされます。

`<Qd>, <Qn>, <Qm>` クワッドワード演算の場合のデスティネーションベクタとオペランドベクタ

`<Dd>, <Dn>, <Dm>` ダブルワード演算の場合のデスティネーションベクタとオペランドベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !IsZero(Elem[D[n+r],e,esize] AND Elem[D[m+r],e,esize]) then
                Elem[D[d+r],e,esize] = Ones(esize);
            else
                Elem[D[d+r],e,esize] = Zeros(esize);
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.409 VUZP

Vector Unzip de-interleaves the elements of two vectors. See Table A8-13 and Table A8-14 for examples of the operation.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

Encoding T1 / A1 Advanced SIMD

```
VUZP<c>.<size>  <Qd>, <Qm>
VUZP<c>.<size>  <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	0	Q	M	0	Vm			

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Table A8-13 Operation of doubleword VUZP.8

	Register state before operation								Register state after operation							
Dd	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	B_6	B_4	B_2	B_0	A_6	A_4	A_2	A_0
Dm	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	B_7	B_5	B_3	B_1	A_7	A_5	A_3	A_1

Table A8-14 Operation of quadword VUZP.32

	Register state before operation				Register state after operation			
Qd	A_3	A_2	A_1	A_0	B_2	B_0	A_2	A_0
Qm	B_3	B_2	B_1	B_0	B_3	B_1	A_3	A_1

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.409 VUZP

VUZP（ベクタアンジップ）は、2 つのベクタのエレメントをインターリーブ解除します。動作の例を表 A8- 13 および表 A8- 14 に示します。

ベクタのエレメントには、8 ビット、16 ビット、または 32 ビットを使用できます。これらにはデータ型の区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VUZP<c>.<size>  <Qd>, <Qm>
VUZP<c>.<size>  <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	0	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	0	Q	M	0	Vm			

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

表 A8- 13 ダブルワード VUZP.8 の動作

	動作前のレジスタの状態								動作後のレジスタの状態							
Dd	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	B_6	B_4	B_2	B_0	A_6	A_4	A_2	A_0
Dm	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	B_7	B_5	B_3	B_1	A_7	A_5	A_3	A_1

表 A8- 14 クワッドワード VUZP.32 の動作

	動作前のレジスタの状態				動作後のレジスタの状態			
Qd	A_3	A_2	A_1	A_0	B_2	B_0	A_2	A_0
Qm	B_3	B_2	B_1	B_0	B_3	B_1	A_3	A_1

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VUZP<c><q>.<size>  <Qd>, <Qm>`	Encoded as Q = 1
`VUZP<c><q>.<size>  <Dd>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VUZP` instruction must be unconditional.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` encoded as size = 0b00.
- `16` encoded as size = 0b01.
- `32` encoded as size = 0b10 for a quadword operation.
  Doubleword operation with `<size>` = 32 is a pseudo-instruction.

`<Qd>, <Qm>` The vectors for a quadword operation.

`<Dd>, <Dm>` The vectors for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN;  Q[m>>1] = bits(128) UNKNOWN;
        else
            zipped_q = Q[m>>1]:Q[d>>1];
            for e = 0 to (128 DIV esize) - 1
                Elem[Q[d>>1],e,esize] = Elem[zipped_q,2*e,esize];
                Elem[Q[m>>1],e,esize] = Elem[zipped_q,2*e+1,esize];
    else
        if d == m then
            D[d] = bits(64) UNKNOWN;  D[m] = bits(64) UNKNOWN;
        else
            zipped_d = D[m]:D[d];
            for e = 0 to (64 DIV esize) - 1
                Elem[D[d],e,esize] = Elem[zipped_d,2*e,esize];
                Elem[D[m],e,esize] = Elem[zipped_d,2*e+1,esize];
```

Exceptions

Undefined Instruction.

Pseudo-instruction

`VUZP.32 <Dd>, <Dm>` is a synonym for `VTRN.32 <Dd>, <Dm>`.

For details see *VTRN* on page A8-800.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

`VUZP<c><q>.<size>  <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VUZP<c><q>.<size>  <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VUZP` は常に無条件命令です。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` クワッドワード演算では size = 0b10 としてエンコードされます。
  `<size>` = 32 の場合のダブルワード演算は擬似命令です。

`<Qd>, <Qm>` クワッドワード演算の場合のベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN;  Q[m>>1] = bits(128) UNKNOWN;
        else
            zipped_q = Q[m>>1]:Q[d>>1];
            for e = 0 to (128 DIV esize) - 1
                Elem[Q[d>>1],e,esize] = Elem[zipped_q,2*e,esize];
                Elem[Q[m>>1],e,esize] = Elem[zipped_q,2*e+1,esize];
    else
        if d == m then
            D[d] = bits(64) UNKNOWN;  D[m] = bits(64) UNKNOWN;
        else
            zipped_d = D[m]:D[d];
            for e = 0 to (64 DIV esize) - 1
                Elem[D[d],e,esize] = Elem[zipped_d,2*e,esize];
                Elem[D[m],e,esize] = Elem[zipped_d,2*e+1,esize];
```

例外

未定義命令

擬似命令

`VUZP.32 <Dd>, <Dm>` は、`VTRN.32 <Dd>, <Dm>` と同義語です。

詳細については、*VTRN* P. A8-800 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.410 VZIP

Vector Zip interleaves the elements of two vectors. See Table A8-15 and Table A8-16 for examples of the operation.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

Encoding T1 / A1 Advanced SIMD

```
VZIP<c>.<size>  <Qd>, <Qm>
VZIP<c>.<size>  <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	1	Q	M	0	Vm			

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Table A8-15 Operation of doubleword VZIP.8

	Register state before operation								Register state after operation							
Dd	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	B_3	A_3	B_2	A_2	B_1	A_1	B_0	A_0
Dm	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	B_7	A_7	B_6	A_6	B_5	A_5	B_4	A_4

Table A8-16 Operation of quadword VZIP.32

	Register state before operation				Register state after operation			
Qd	A_3	A_2	A_1	A_0	B_1	A_1	B_0	A_0
Qm	B_3	B_2	B_1	B_0	B_3	A_3	B_2	A_2

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.410 VZIP

VZIP（ベクタジップ）は、2 つのベクタのエレメントをインターリーブします。動作の例を表 A8- 15 および表 A8- 16 に示します。

ベクタのエレメントには、8 ビット、16 ビット、または 32 ビットを使用できます。これらにはデータ型の区別はありません。

エンコード T1/A1 アドバンスト SIMD

```
VZIP<c>.<size>  <Qd>, <Qm>
VZIP<c>.<size>  <Dd>, <Dm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	1	Q	M	0	Vm			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	1	Q	M	0	Vm			

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

表 A8- 15 ダブルワード VZIP.8 の動作

	動作前のレジスタの状態								動作後のレジスタ状態							
Dd	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	B_3	A_3	B_2	A_2	B_1	A_1	B_0	A_0
Dm	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	B_7	A_7	B_6	A_6	B_5	A_5	B_4	A_4

表 A8- 16 クワッドワード VZIP.32 の動作

	動作前のレジスタの状態				動作後のレジスタの状態			
Qd	A_3	A_2	A_1	A_0	B_1	A_1	B_0	A_0
Qm	B_3	B_2	B_1	B_0	B_3	A_3	B_2	A_2

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

`VZIP<c><q>.<size>  <Qd>, <Qm>`	Encoded as Q = 1
`VZIP<c><q>.<size>  <Dd>, <Dm>`	Encoded as Q = 0

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `VZIP` instruction must be unconditional.

`<size>` The data size for the elements of the vectors. It must be one of:

- `8` encoded as size = 0b00.
- `16` encoded as size = 0b01.
- `32` encoded as size = 0b10 for a quadword operation.
  Doubleword operation with `<size>` = 32 is a pseudo-instruction.

`<Qd>, <Qm>` The vectors for a quadword operation.

`<Dd>, <Dm>` The vectors for a doubleword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN;  Q[m>>1] = bits(128) UNKNOWN;
        else
            bits(256) zipped_q;
            for e = 0 to (128 DIV esize) - 1
                Elem[zipped_q,2*e,esize] = Elem[Q[d>>1],e,esize];
                Elem[zipped_q,2*e+1,esize] = Elem[Q[m>>1],e,esize];
            Q[d>>1] = zipped_q<127:0>;  Q[m>>1] = zipped_q<255:128>;
    else
        if d == m then
            D[d] = bits(64) UNKNOWN;  D[m] = bits(64) UNKNOWN;
        else
            bits(128) zipped_d;
            for e = 0 to (64 DIV esize) - 1
                Elem[zipped_d,2*e,esize] = Elem[D[d],e,esize];
                Elem[zipped_d,2*e+1,esize] = Elem[D[m],e,esize];
            D[d] = zipped_d<63:0>;  D[m] = zipped_d<127:64>;
```

Exceptions

Undefined Instruction.

Pseudo-instructions

`VZIP.32 <Dd>, <Dm>` is a synonym for `VTRN.32 <Dd>, <Dm>`.

For details see *VTRN* on page A8-800.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

`VZIP<c><q>.<size>  <Qd>, <Qm>`	Q = 1 としてエンコードされる
`VZIP<c><q>.<size>  <Dd>, <Dm>`	Q = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `VZIP` は常に無条件命令です。

`<size>` ベクタのエレメントのデータサイズ。次のいずれかを使用できます。

- `8` size = 0b00 としてエンコードされます。
- `16` size = 0b01 としてエンコードされます。
- `32` クワッドワード演算では size = 0b10 としてエンコードされます。
  `<size>` = 32 の場合のダブルワード演算は擬似命令です。

`<Qd>, <Qm>` クワッドワード演算の場合のベクタ

`<Dd>, <Dm>` ダブルワード演算の場合のベクタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN;  Q[m>>1] = bits(128) UNKNOWN;
        else
            bits(256) zipped_q;
            for e = 0 to (128 DIV esize) - 1
                Elem[zipped_q,2*e,esize] = Elem[Q[d>>1],e,esize];
                Elem[zipped_q,2*e+1,esize] = Elem[Q[m>>1],e,esize];
            Q[d>>1] = zipped_q<127:0>;  Q[m>>1] = zipped_q<255:128>;
    else
        if d == m then
            D[d] = bits(64) UNKNOWN;  D[m] = bits(64) UNKNOWN;
        else
            bits(128) zipped_d;
            for e = 0 to (64 DIV esize) - 1
                Elem[zipped_d,2*e,esize] = Elem[D[d],e,esize];
                Elem[zipped_d,2*e+1,esize] = Elem[D[m],e,esize];
            D[d] = zipped_d<63:0>;  D[m] = zipped_d<127:64>;
```

例外

未定義命令

擬似命令

`VZIP.32 <Dd>, <Dm>` は、`VTRN.32 <Dd>, <Dm>` と同義語です。

詳細については、*VTRN* P. A8-800 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.411 WFE

Wait For Event is a hint instruction that permits the processor to enter a low-power state until one of a number of events occurs, including events signaled by executing the SEV instruction on any processor in the multiprocessor system. For more information, see *Wait For Event and Send Event* on page B1-44.

Encoding T1 ARMv7 (executes as NOP in ARMv6T2)

```
WFE<c>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0

```
// No additional decoding required
```

Encoding T2 ARMv7 (executes as NOP in ARMv6T2)

```
WFE<c>.W
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	0

```
// No additional decoding required
```

Encoding A1 ARMv6K, ARMv7 (executes as NOP in ARMv6T2)

```
WFE<c>
```

31 30 29 28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	1	0

```
// No additional decoding required
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.411 WFE

WFE（イベント待ち）は、複数のイベントのいずれかが発生するまで、プロセッサが低電力状態に遷移することを許可するヒント命令です。これらのイベントには、マルチプロセッサシステムの任意のプロセッサに対して SEV 命令を実行することによって伝達されるイベントが含まれます。詳細については、*イベント待ちとイベント送信* P. B1-44 を参照して下さい。

エンコード T1 ARMv7（ARMv6T2 では NOP として動作する）

```
WFE<c>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0

```
// No additional decoding required
```

エンコード T2 ARMv7（ARMv6T2 では NOP として動作する）

```
WFE<c>.W
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	0

```
// No additional decoding required
```

エンコード A1 ARMv6K、ARMv7（ARMv6T2 では NOP として動作する）

```
WFE<c>
```

31 30 29 28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond	0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	1	0

```
// No additional decoding required
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
WFE<c><q>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if EventRegistered() then
        ClearEventRegister();
    else
        WaitForEvent();
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
WFE<c><q>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if EventRegistered() then
        ClearEventRegister();
    else
        WaitForEvent();
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.412 WFI

Wait For Interrupt is a hint instruction that permits the processor to enter a low-power state until one of a number of asynchronous events occurs. For details, see *Wait For Interrupt* on page B1-47.

Encoding T1 ARMv7 (executes as NOP in ARMv6T2)

```
WFI<c>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	1	0	0	0	0

```
// No additional decoding required
```

Encoding T2 ARMv7 (executes as NOP in ARMv6T2)

```
WFI<c>.W
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	1

```
// No additional decoding required
```

Encoding A1 ARMv6K, ARMv7 (executes as NOP in ARMv6T2)

```
WFI<c>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	1	1

```
// No additional decoding required
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.412 WFI

WFI（割り込み待ち）は、複数の非同期イベントのいずれかが発生するまで、プロセッサが低電力状態に遷移することを許可するヒント命令です。詳細については、*割り込み待ち* P. B1- 47 を参照して下さい。

エンコード T1 ARMv7（ARMv6T2 では NOP として動作する）

```
WFI<c>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	1	0	0	0	0

```
// No additional decoding required
```

エンコード T2 ARMv7（ARMv6T2 では NOP として動作する）

```
WFI<c>.W
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	1

```
// No additional decoding required
```

エンコード A1 ARMv6K、ARMv7（ARMv6T2 では NOP として動作する）

```
WFI<c>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	1	1

```
// No additional decoding required
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
WFI<c><q>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    WaitForInterrupt();
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
WFI<c><q>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    WaitForInterrupt();
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A8.6.413 YIELD

YIELD is a hint instruction. It enables software with a multithreading capability to indicate to the hardware that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. Hardware can use this hint to suspend and resume multiple code threads if it supports the capability.

Encoding T1 ARMv7 (executes as NOP in ARMv6T2)

YIELD<c>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0

```
// No additional decoding required
```

Encoding T2 ARMv7 (executes as NOP in ARMv6T2)

YIELD<c>.W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	1

```
// No additional decoding required
```

Encoding A1 ARMv6K, ARMv7 (executes as NOP in ARMv6T2)

YIELD<c>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond					0	0	1	1	0	0	1	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	0	1

```
// No additional decoding required
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A8.6.413 YIELD

YIELD（イールド）はヒント命令です。マルチスレッド機能を持つソフトウェアがハードウェアに対して、スピンロックなど、システム全体の性能向上のためにスワップアウト可能なタスクを実行中であることを通知することができます。ハードウェアは、このヒントを使用して、複数のコードスレッドを中断および再開させることができます（この機能をサポートしている場合）。

エンコード T1 ARMv7（ARMv6T2 では NOP として動作する）

YIELD<c>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0

```
// No additional decoding required
```

エンコード T2 ARMv7（ARMv6T2 では NOP として動作する）

YIELD<c>.W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	1

```
// No additional decoding required
```

エンコード A1 ARMv6K、ARMv7（ARMv6T2 では NOP として動作する）

YIELD<c>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond					0	0	1	1	0	0	1	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	0	1

```
// No additional decoding required
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
YIELD<c><q>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
YIELD<c><q>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter A9
ThumbEE

This chapter contains detailed reference material on ThumbEE.

It contains the following sections:

- *The ThumbEE instruction set* on page A9-2
- *ThumbEE instruction set encoding* on page A9-6
- *Additional instructions in Thumb and ThumbEE instruction sets* on page A9-7
- *ThumbEE instructions with modified behavior* on page A9-8
- *Additional ThumbEE instructions* on page A9-14.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 A9 章
ThumbEE

本章では、ThumbEE に関する詳細な参照資料を記載します。

本章は次のセクションから構成されています。

- *ThumbEE 命令セット* P. A9-2
- *ThumbEE 命令セットのエンコード* P. A9-6
- *Thumb 命令セットおよび ThumbEE 命令セットの追加命令* P. A9-7
- *動作が変更された ThumbEE 命令* P. A9-8
- *追加の ThumbEE 命令* P. A9-14

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.1 The ThumbEE instruction set

In general, instructions in ThumbEE are identical to Thumb instructions, with the following exceptions:

- A small number of instructions are affected by modifications to transitions from ThumbEE state. For more information, see *ThumbEE state transitions*.
- A substantial number of instructions have a null check on the base register before any other operation takes place, but are identical (or almost identical) in all other respects. For more information, see *Null checking* on page A9-3.
- A small number of instructions are modified in additional ways. See *Instructions with modifications* on page A9-4.
- Three Thumb instructions, BLX (immediate), 16-bit LDM, and 16-bit STM, are removed in ThumbEE state.

  The encoding corresponding to BLX (immediate) in Thumb is UNDEFINED in ThumbEE state.

  16-bit LDM and STM are replaced by new instructions, for details see *Additional ThumbEE instructions* on page A9-14.
- Two new 32-bit instructions, ENTERX and LEAVEX, are introduced in both the Thumb instruction set and the ThumbEE instruction set. See *Additional instructions in Thumb and ThumbEE instruction sets* on page A9-7. These instructions use previously UNDEFINED encodings.

A9.1.1 ThumbEE state transitions

Instruction set state transitions to ThumbEE state can occur implicitly as part of a return from exception, or explicitly on execution of an ENTERX instruction.

Instruction set state transitions from ThumbEE state can only occur due to an exception, or due to a transition to Thumb state using the LEAVEX instruction. Return from exception instructions (RFE and SUBS PC, LR, #imm) are UNPREDICTABLE in ThumbEE state.

Any other Thumb instructions that can update the PC in ThumbEE state are UNPREDICTABLE if they attempt to change to ARM state. Interworking of ARM and Thumb instructions is not supported in ThumbEE state. The instructions affected are:

- LDR, LDM, and POP instructions that write to the PC, if bit [0] of the value loaded to the PC is 0
- BLX (register), BX, and BXJ, where Rm bit [0] == 0.

Note

SVC, BKPT, and UNDEFINED instructions cause an exception to occur.

If a BXJ <Rm> instruction is executed in ThumbEE state, with Rm bit[0] == 1, it does not enter Jazelle state. Instead, it behaves like the corresponding BX <Rm> instruction and remains in ThumbEE state.

Debug state is a special case. For the rules governing changes to CPSR state bits and Debug state, see *Executing instructions in Debug state* on page C5-9.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.1 ThumbEE 命令セット

一般に、ThumbEE に含まれる命令は Thumb 命令と同じですが、次の点が異なります。

- 一部の命令は、ThumbEE 状態からの遷移に対する変更によって影響を受けます。詳細については、*ThumbEE 状態への遷移*を参照して下さい。
- かなりの数の命令で、他の動作が行われる前に、ベースレジスタに対する NULL チェックが実行されますが、その他の点では同じ（またはほとんど同じ）です。詳細については、*NULL チェック* P. A9-3 を参照して下さい。
- 一部の命令は、他の部分で変更されています。詳細については、*変更を伴う命令* P. A9-4 を参照して下さい。
- BLX（イミディエート）、16 ビット LDM、16 ビット STM の 3 つの Thumb 命令は、ThumbEE 状態では削除されています。

  Thumb の BLX（イミディエート）に対応するエンコードは、ThumbEE 状態では未定義です。

  16 ビット LDM および STM は、新しい命令に置き換えられます。詳細については、*追加の ThumbEE 命令* P. A9-14 を参照して下さい。
- ENTERX および LEAVEX という 2 つの新しい 32 ビット命令が、Thumb 命令セットおよび ThumbEE 命令セットの両方に導入されています。詳細については、*Thumb 命令セットおよび ThumbEE 命令セットの追加命令* P. A9-7 を参照して下さい。これらの命令では、これまで未定義であったエンコードが使用されます。

A9.1.1 ThumbEE 状態への遷移

ThumbEE 状態への命令セット状態遷移は、例外からの復帰の一部として暗黙的に発生するか、ENTERX 命令の実行時に明示的に発生します。

ThumbEE 状態からの命令セット状態遷移は、例外の結果として、または LEAVEX 命令を使用した Thumb 状態への遷移の結果としてのみ発生します。例外命令からの復帰（RFE および SUBS PC, LR, #imm）は、ThumbEE 状態では予測不能です。

ThumbEE 状態で PC を更新できる他の Thumb 命令はすべて、ARM 状態への変更を試みる場合、予測不能です。ARM 命令および Thumb 命令のインターワーキングは、ThumbEE 状態ではサポートされません。影響を受ける命令は次のとおりです。

- PC にロードされる値のビット [0] が 0 の場合に PC に書き込む、LDR、LDM、POP 命令
- BLX（レジスタ）、BX、および BXJ で Rm のビット [0] == 0 の場合

注

SVC、BKPT、および未定義命令は例外を引き起こします。

BXJ <Rm> 命令（Rm のビット [0] == 1）が ThumbEE 状態で実行される場合、Jazelle 状態には入りません。その代わり、対応する BX <Rm> 命令のように動作し、ThumbEE 状態に留まります。

デバッグ状態は特殊なケースです。CPSR 状態ビットおよびデバッグ状態への変更に適用される規則については、*デバッグ状態での命令の実行* P. C5-9 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.1.2 Null checking

A *null check* is performed for all load/store instructions when they are executed in ThumbEE state. If the value in the base register is zero, execution branches to the NullCheck handler at HandlerBase – 4.

For most load/store instructions, this is the only difference from normal Thumb operation. Exceptions to this rule are described in this chapter.

Note

- The null check examines the value in the base register, not any calculated value offset from the base register.
- If the base register is the SP or the PC, a zero value in the base register results in UNPREDICTABLE behavior.

The instructions affected by null checking are:

- all instructions whose mnemonic starts with LD, ST, VLD or VST
- POP, PUSH, TBB, TBH, VPOP, and VPUSH.

For each of these instructions, the pseudocode shown in the Operation section uses the following function:

```
// NullCheckIfThumbEE()
// ====================

NullCheckIfThumbEE(integer n)
    if CurrentInstrSet() == InstrSet_ThumbEE then
        if n == 15 then
            if IsZero(Align(PC,4)) then UNPREDICTABLE;
        elsif n == 13 then
            if IsZero(SP) then UNPREDICTABLE;
        else
            if IsZero(R[n]) then
                LR = PC<31:1> : '1';  // PC holds this instruction's address plus 4
                BranchWritePC(TEEHBR - 4);
                EndOfInstruction();
    return;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.1.2 NULL チェック

*NULL チェック*は、ThumbEE 状態で実行されるすべてのロード / ストア命令に対して実行されます。ベースレジスタの値が 0 の場合、HandlerBase - 4 において実行が NullCheck ハンドラに分岐します。

大半のロード / ストア命令では、これが通常の Thumb 動作と異なる唯一の点です。この規則の例外については、本章で説明します。

注

- NULL チェックでは、ベースレジスタの値を調べますが、ベースレジスタからの計算済みオフセット値は調べません。
- ベースレジスタが SP または PC の場合、ベースレジスタの値が 0 であると、動作は予測不能です。

NULL チェックの影響を受ける命令は次のとおりです。

- ニーモニックが LD、ST、VLD、VST で始まるすべての命令
- POP、PUSH、TBB、TBH、VPOP、VPUSH

これらの命令すべてについて、動作セクションに示されている擬似コードで、次の関数が使用されます。

```
// NullCheckIfThumbEE()
// ====================

NullCheckIfThumbEE(integer n)
    if CurrentInstrSet() == InstrSet_ThumbEE then
        if n == 15 then
            if IsZero(Align(PC,4)) then UNPREDICTABLE;
        elsif n == 13 then
            if IsZero(SP) then UNPREDICTABLE;
        else
            if IsZero(R[n]) then
                LR = PC<31:1> : '1';  // PC holds this instruction's address plus 4
                BranchWritePC(TEEHBR - 4);
                EndOfInstruction();
    return;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.1.3 Instructions with modifications

In addition to the instructions described in *ThumbEE state transitions* on page A9-2 and *Null checking* on page A9-3, Table A9-1 shows other instructions that are modified in ThumbEE state. The pseudocode, including the null check if any, is given in *ThumbEE instructions with modified behavior* on page A9-8.

Table A9-1 Modified instructions

Instructions	Rbase	Modification
LDR (register)	Rn	Rm multiplied by 4, null check
LDRH (register)	Rn	Rm multiplied by 2, null check
LDRSH (register)	Rn	Rm multiplied by 2, null check
STR (register)	Rn	Rm multiplied by 4, null check
STRH (register)	Rn	Rm multiplied by 2, null check

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.1.3 変更を伴う命令

ThumbEE 状態への遷移 P. A9- 2 および *NULL チェック* P. A9- 3 で説明した命令に加え、ThumbEE 状態で変更されるその他の命令を表 A9- 1 に示します。NULL チェックを含む擬似コードについては、*動作が変更された ThumbEE 命令* P. A9- 8 を参照して下さい。

表 A9- 1 変更される命令

命令	Rbase	変更内容
LDR（レジスタ）	Rn	Rm × 4、NULL チェック
LDRH（レジスタ）	Rn	Rm × 2、NULL チェック
LDRSH（レジスタ）	Rn	Rm × 2、NULL チェック
STR（レジスタ）	Rn	Rm × 4、NULL チェック
STRH（レジスタ）	Rn	Rm × 2、NULL チェック

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.1.4 IT block and check handlers

CHKA, stores, and permitted loads (loads to the PC are only permitted as the last instruction) can occur anywhere in an IT block. If one of these instructions results in a branch to the null pointer or array index handlers, the IT state bits in ITSTATE are cleared. This enables unconditional execution from the start of the handler.

The original IT state bits are not preserved.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.1.4 IT ブロックおよびチェックハンドラ

CHKA 命令、ストア命令、および許可されたロード命令（PC へのロードは最後の命令としてのみ許可されます）は、IT ブロックの任意の場所で実行できます。これらの命令のいずれかによって、NULL ポインタまたはアレイインデクス ハンドラへの分岐が発生した場合、ITSTATE の IT 状態ビットがクリアされます。これにより、ハンドラの先頭からの無条件実行が可能になります。

元の IT 状態ビットは保持されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.2 ThumbEE instruction set encoding

In general, instructions in the ThumbEE instruction set are encoded in exactly the same way as Thumb instructions described in Chapter A6 *Thumb Instruction Set Encoding*. The differences are as follows:

- There are no 16-bit LDM or STM instructions in the ThumbEE instruction set.
- The 16-bit encodings used for LDM and STM in the Thumb instruction set are used for different 16-bit instructions in the ThumbEE instruction set. For details, see *16-bit ThumbEE instructions*.
- There are two new 32-bit instructions in both Thumb state and ThumbEE state. For details, see *Additional instructions in Thumb and ThumbEE instruction sets* on page A9-7.

A9.2.1 16-bit ThumbEE instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Opcode											

Table A9-2 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A9-2 16-bit ThumbEE instructions

Opcode	Instruction	See
0000	Handler Branch with Parameter	*HBP* on page A9-18
0001	UNDEFINED	
001x	Handler Branch, Handler Branch with Link	*HB, HBL* on page A9-16
01xx	Handler Branch with Link and Parameter	*HBLP* on page A9-17
100x	Load Register from a frame	*LDR (immediate)* on page A9-19
1010	Check Array	*CHKA* on page A9-15
1011	Load Register from a literal pool	*LDR (immediate)* on page A9-19
110x	Load Register (array operations)	*LDR (immediate)* on page A9-19
111x	Store Register to a frame	*STR (immediate)* on page A9-21

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.2 ThumbEE 命令セットのエンコード

一般に、ThumbEE 命令セットに含まれる命令は、第 A6 章*「Thumb 命令セットのエンコード」*に記載されている Thumb 命令と正確に同じ方法でエンコードされます。ただし、次のような違いがあります。

- ThumbEE 命令セットには、16 ビット LDM 命令および STM 命令がありません。
- Thumb 命令セットの LDM および STM で使用される 16 ビットエンコードは、ThumbEE 命令セットの別の 16 ビット命令で使用されます。詳細については、*16 ビット ThumbEE 命令*を参照して下さい。
- Thumb 状態と ThumbEE 状態の両方に、新しい 32 ビット命令が 2 つ存在します。詳細については、*Thumb 命令セットおよび ThumbEE 命令セットの追加命令* P. A9-7 を参照して下さい。

A9.2.1 16 ビット ThumbEE 命令

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Opcode											

この空間でのエンコードの割り当てを、表 A9-2 に示します。この空間に含まれる他のエンコードは未定義です。

表 A9-2 16 ビット ThumbEE 命令

Opcode	命令	参照先
0000	パラメータ付きハンドラ分岐	*HBP* P. A9-18
0001	未定義	
001x	ハンドラ分岐、リンク付きハンドラ分岐	*HB*、*HBL* P. A9-16
01xx	リンク / パラメータ付きハンドラ分岐	*HBLP* P. A9-17
100x	フレームからのレジスタロード	*LDR（イミディエート）* P. A9-19
1010	配列チェック	*CHKA* P. A9-15
1011	リテラルプールからのレジスタロード	*LDR（イミディエート）* P. A9-19
110x	レジスタロード（配列操作）	*LDR（イミディエート）* P. A9-19
111x	フレームへのレジスタストア	*STR（イミディエート）* P. A9-21

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.3 Additional instructions in Thumb and ThumbEE instruction sets

On a processor with the ThumbEE extension, there are two additional 32-bit instructions, ENTERX and LEAVEX. These are available in both Thumb state and ThumbEE state.

A9.3.1 ENTERX, LEAVEX

ENTERX causes a change from Thumb state to ThumbEE state, or has no effect in ThumbEE state.

LEAVEX causes a change from ThumbEE state to Thumb state, or has no effect in Thumb state.

Encoding T1 ThumbEE

ENTERX	Not permitted in IT block.
LEAVEX	Not permitted in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	0	0	J	(1)	(1)	(1)	(1)

```
is_enterx = (J == '1');
```

Assembler syntax

ENTERX<q>	Encoded as J = 1
LEAVEX<q>	Encoded as J = 0

where:

<q> See *Standard assembler syntax fields* on page A8-7. An ENTERX or LEAVEX instruction must be unconditional.

Operation

```
if is_enterx then
    SelectInstrSet(InstrSet_ThumbEE);
else
    SelectInstrSet(InstrSet_Thumb);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.3 Thumb 命令セットおよび ThumbEE 命令セットの追加命令

ThumbEE 拡張機能を持つプロセッサには、ENTERX および LEAVEX という 2 つの 32 ビット命令が追加されています。これらの命令は、Thumb 状態と ThumbEE 状態の両方で使用できます。

A9.3.1 ENTERX、LEAVEX

ENTERX は Thumb 状態から ThumbEE 状態への変更を引き起こしますが、ThumbEE 状態では何の影響もありません。

LEAVEX は ThumbEE 状態から Thumb 状態への変更を引き起こしますが、Thumb 状態では何の影響もありません。

エンコード T1 ThumbEE

ENTERX	IT ブロック内部では許可されない
LEAVEX	IT ブロック内部では許可されない

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	0	0	J	(1)	(1)	(1)	(1)

```
is_enterx = (J == '1');
```

アセンブラ構文

ENTERX<q>	J = 1 としてエンコードされる
LEAVEX<q>	J = 0 としてエンコードされる

各項目の説明については以下を参照して下さい。

<q> *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。ENTERX および LEAVEX は常に無条件命令です。

動作

```
if is_enterx then
    SelectInstrSet(InstrSet_ThumbEE);
else
    SelectInstrSet(InstrSet_Thumb);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.4 ThumbEE instructions with modified behavior

The 16-bit encodings of the following Thumb instructions have changed functionality in ThumbEE:

- *LDR (register)* on page A9-9
- *LDRH (register)* on page A9-10
- *LDRSH (register)* on page A9-11
- *STR (register)* on page A9-12
- *STRH (register)* on page A9-13.

In ThumbEE state there are the following changes in the behavior of instructions:

- All load/store instructions perform null checks on their base register values, as described in *Null checking* on page A9-3. The pseudocode for these instructions in Chapter A8 *Instruction Details* describes this by calling the `NullCheckIfThumbEE()` pseudocode procedure.
- Instructions that attempt to enter ARM state are UNPREDICTABLE, as described in *ThumbEE state transitions* on page A9-2. The pseudocode for these instructions in Chapter A8 *Instruction Details* describes this by calling the `SelectInstrSet()` or `BXWritePC()` pseudocode procedure.
- The `BXJ` instruction behaves like the `BX` instruction, as described in *ThumbEE state transitions* on page A9-2. The pseudocode for the instruction, in *BXJ* on page A8-64, describes this directly.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.4 動作が変更された ThumbEE 命令

次の Thumb 命令の 16 ビットエンコードは、ThumbEE で機能が変更されています。

- *LDR（レジスタ）* P. A9- 9
- *LDRH（レジスタ）* P. A9- 10
- *LDRSH（レジスタ）* P. A9- 11
- *STR（レジスタ）* P. A9- 12
- *STRH（レジスタ）* P. A9- 13

ThumbEE 状態では、命令の動作に次のような変更があります。

- *NULL チェック* P. A9- 3 で説明されているように、すべてのロード / ストア命令が、ベースレジスタの値に対する NULL チェックを実行します。第 A8 章 *「命令の詳細」* に示されているこれらの命令の擬似コードは、`NullCheckIfThumbEE()` 擬似コードプロシージャを呼び出すことで、この動作を記述します。
- *ThumbEE 状態への遷移* P. A9- 2 で説明されているように、ARM 状態の開始を試みる命令は予測不能です。第 A8 章 *「命令の詳細」* に示されているこれらの命令の擬似コードは、`SelectInstrSet()` または `BXWritePC()` 擬似コードプロシージャを呼び出すことで、この動作を記述します。
- *ThumbEE 状態への遷移* P. A9- 2 で説明されているように、`BXJ` 命令は `BX` 命令と同様に動作します。*BXJ* P. A8- 64 に示されているこの命令の擬似コードは、この動作を直接記述します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.4.1 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value is shifted left by 2 bits. For information about memory accesses see *Memory accesses* on page A8-13.

The similar Thumb instruction does not have a left shift.

Encoding T1 ThumbEE

```
LDR<c> <Rt>,[<Rn>,<Rm>, LSL #2]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	1 0 0	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
```

Assembler syntax

```
LDR<c><q>  <Rt>, [<Rn>, <Rm>, LSL #2]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register.

`<Rm>` Contains the offset that is shifted and applied to the value of `<Rn>` to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + LSL(R[m],2);
    R[t] = MemU[address,4];
```

Exceptions and checks

Data Abort, NullCheck.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.4.1 LDR（レジスタ）

LDR（レジスタロード（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 ワードをロードし、レジスタに書き込みます。オフセットレジスタの値は、左に 2 ビットシフトされます。メモリアクセスの詳細については、*メモリアクセス*P. A8- 13 を参照して下さい。

これに類似する Thumb 命令では、左シフトは適用されません。

エンコード T1 ThumbEE

```
LDR<c> <Rt>,[<Rn>,<Rm>, LSL #2]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	1 0 0	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
```

アセンブラ構文

```
LDR<c><q>  <Rt>, [<Rn>, <Rm>, LSL #2]
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ

`<Rm>` アドレス生成のためにシフトされて、`<Rn>` の値に適用されるオフセット

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + LSL(R[m],2);
    R[t] = MemU[address,4];
```

例外およびチェック

データアボート、NULL チェック

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.4.2 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value is shifted left by 1 bit. For information about memory accesses see *Memory accesses* on page A8-13.

The similar Thumb instruction does not have a left shift.

Encoding T1 ThumbEE

```
LDRH<c> <Rt>,[<Rn>,<Rm>, LSL #1]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	1 0 1	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
```

Assembler syntax

```
LDRH<c><q>  <Rt>, [<Rn>, <Rm>, LSL #1]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register.

`<Rm>` Contains the offset that is shifted and applied to the value of `<Rn>` to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + LSL(R[m],1);
    R[t] = ZeroExtend(MemU[address,2], 32);
```

Exceptions and checks

Data Abort, NullCheck.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.4.2 LDRH（レジスタ）

LDRH（レジスタロード ハーフワード（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。オフセットレジスタの値は、左に 1 ビットシフトされます。メモリアクセスの詳細については、*メモリアクセス*P. A8-13 を参照して下さい。

これに類似する Thumb 命令では、左シフトは適用されません。

エンコード T1 ThumbEE

```
LDRH<c> <Rt>,[<Rn>,<Rm>, LSL #1]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	1 0 1	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
```

アセンブラ構文

```
LDRH<c><q>  <Rt>, [<Rn>, <Rm>, LSL #1]
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ

`<Rm>` アドレス生成のためにシフトされて、`<Rn>` の値に適用されるオフセット

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + LSL(R[m],1);
    R[t] = ZeroExtend(MemU[address,2], 32);
```

例外およびチェック

データアボート、NULL チェック

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.4.3 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value is shifted left by 1 bit. For information about memory accesses see *Memory accesses* on page A8-13.

The similar Thumb instruction does not have a left shift.

Encoding T1 ThumbEE

```
LDRSH<c> <Rt>,[<Rn>,<Rm>, LSL #1]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	1 1 1	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
```

Assembler syntax

```
LDRSH<c><q>  <Rt>, [<Rn>, <Rm>, LSL #1]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register.

`<Rm>` Contains the offset that is shifted and applied to the value of `<Rn>` to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + LSL(R[m],1);
    R[t] = SignExtend(MemU[address,2], 32);
```

Exceptions and checks

Data Abort, NullCheck.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.4.3 LDRSH（レジスタ）

LDRSH（レジスタロード符号付きハーフワード（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。オフセットレジスタの値は、左に 1 ビットシフトされます。メモリアクセスの詳細については、*メモリアクセス* P. A8-13 を参照して下さい。

これに類似する Thumb 命令では、左シフトは適用されません。

エンコード T1 ThumbEE

```
LDRSH<c> <Rt>,[<Rn>,<Rm>, LSL #1]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	1 1 1	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
```

アセンブラ構文

```
LDRSH<c><q>  <Rt>, [<Rn>, <Rm>, LSL #1]
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ

`<Rm>` アドレス生成のためにシフトされて、`<Rn>` の値に適用されるオフセット

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + LSL(R[m],1);
    R[t] = SignExtend(MemU[address,2], 32);
```

例外およびチェック

データアボート、NULL チェック

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.4.4 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a word from a register to memory. The offset register value is shifted left by 2 bits. For information about memory accesses see *Memory accesses* on page A8-13.

The similar Thumb instruction does not have a left shift.

Encoding T1 ThumbEE

```
STR<c> <Rt>,[<Rn>,<Rm>, LSL #2]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	0 0 0	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
```

Assembler syntax

```
STR<c><q>  <Rt>, [<Rn>, <Rm>, LSL #2]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register.

`<Rn>` The base register.

`<Rm>` Contains the offset that is shifted and applied to the value of `<Rn>` to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + LSL(R[m],2);
    MemU[address,4] = R[t];
```

Exceptions and checks

Data Abort, NullCheck.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.4.4 STR（レジスタ）

STR（レジスタストア（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、レジスタからそのメモリに 1 ワードをストアします。オフセットレジスタの値は、左に 2 ビットシフトされます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

これに類似する Thumb 命令では、左シフトは適用されません。

エンコード T1 ThumbEE

```
STR<c> <Rt>,[<Rn>,<Rm>, LSL #2]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	0 0 0	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
```

アセンブラ構文

```
STR<c><q>  <Rt>, [<Rn>, <Rm>, LSL #2]
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>` ソースレジスタ

`<Rn>` ベースレジスタ

`<Rm>` アドレス生成のためにシフトされて、`<Rn>` の値に適用されるオフセット

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + LSL(R[m],2);
    MemU[address,4] = R[t];
```

例外およびチェック

データアボート、NULL チェック

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.4.5 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. The offset register value is shifted left by 1 bit. For information about memory accesses see *Memory accesses* on page A8-13.

The similar Thumb instruction does not have a left shift.

Encoding T1 ThumbEE

```
STRH<c> <Rt>,[<Rn>,<Rm>, LSL #1]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	0 0 1	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
```

Assembler syntax

```
STRH<c><q>  <Rt>, [<Rn>, <Rm>, LSL #1]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register.

`<Rn>` The base register.

`<Rm>` Contains the offset that is shifted and applied to the value of `<Rn>` to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + LSL(R[m],1);
    MemU[address,2] = R[t]<15:0>;
```

Exceptions and checks

Data Abort, NullCheck.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.4.5 STRH（レジスタ）

STRH（レジスタストア ハーフワード（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、レジスタからそのメモリに 1 ハーフワードをストアします。オフセットレジスタの値は、左に 1 ビットシフトされます。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

これに類似する Thumb 命令では、左シフトは適用されません。

エンコード T1 ThumbEE

```
STRH<c> <Rt>,[<Rn>,<Rm>, LSL #1]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0 1 0 1	0 0 1	Rm	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
```

アセンブラ構文

```
STRH<c><q>  <Rt>, [<Rn>, <Rm>, LSL #1]
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>` ソースレジスタ

`<Rn>` ベースレジスタ

`<Rm>` アドレス生成のためにシフトされて、`<Rn>` の値に適用されるオフセット

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = R[n] + LSL(R[m],1);
    MemU[address,2] = R[t]<15:0>;
```

例外およびチェック

データアボート、NULL チェック

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.5 Additional ThumbEE instructions

The following instructions are available in ThumbEE state, but not in Thumb state:

- *CHKA* on page A9-15
- *HB, HBL* on page A9-16
- *HBLP* on page A9-17
- *HBP* on page A9-18
- *LDR (immediate)* on page A9-19
- *STR (immediate)* on page A9-21.

These are 16-bit instructions. They occupy the instruction encoding space that `STMIA` and `LDMIA` occupy in Thumb state.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.5 追加の ThumbEE 命令

次の命令は ThumbEE 状態で使用できますが、Thumb 状態では使用できません。

- *CHKA* P. A9- 15
- *HB*、*HBL* P. A9- 16
- *HBLP* P. A9- 17
- *HBP* P. A9- 18
- *LDR（イミディエート）* P. A9- 19
- *STR（イミディエート）* P. A9- 21

これらは 16 ビット命令です。これらの命令は、Thumb 状態で `STMIA` および `LDMIA` が占有する命令エンコード空間を占有します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.5.1 CHKA

CHKA (Check Array) compares the unsigned values in two registers. If the first is lower than, or the same as, the second, it copies the PC to the LR, and causes a branch to the IndexCheck handler.

Encoding E1 ThumbEE

```
CHKA<c> <Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4 3	2 1 0
1 1 0 0	1 0 1 0	N	Rm	Rn

```
n = UInt(N:Rn);  m = UInt(Rm);
if n == 15 || BadReg(m) then UNPREDICTABLE;
```

Assembler syntax

```
CHKA<c><q>  <Rn>, <Rm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<Rn> The first operand register. This contains the array size. Use of the SP is permitted.

<Rm> The second operand register. This contains the array index.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if UInt(R[n]) <= UInt(R[m]) then
        LR = PC<31:1> : '1';  // PC holds this instruction's address + 4
        BranchWritePC(TEEHBR - 8);
```

Exceptions and checks

IndexCheck.

Usage

Use CHKA to check that an array index is in bounds.

CHKA does not modify the APSR condition code flags.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.5.1 CHKA

CHKA（配列チェック）は、2 つのレジスタの符号なしの値を比較します。最初の値が 2 番目の値以下である場合、この命令は PC を LR にコピーし、IndexCheck ハンドラへの分岐を引き起こします。

エンコード E1 ThumbEE

```
CHKA<c> <Rn>,<Rm>
```

15 14 13 12	11 10 9 8	7	6 5 4 3	2 1 0
1 1 0 0	1 0 1 0	N	Rm	Rn

```
n = UInt(N:Rn);  m = UInt(Rm);
if n == 15 || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
CHKA<c><q>  <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<Rn> 最初のオペランドレジスタ。配列のサイズを格納します。SP を使用できます。

<Rm> 2 番目のオペランドレジスタ。配列のインデクスを格納します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if UInt(R[n]) <= UInt(R[m]) then
        LR = PC<31:1> : '1';  // PC holds this instruction's address + 4
        BranchWritePC(TEEHBR - 8);
```

例外およびチェック

IndexCheck

使用法

CHKA を使用して、配列のインデクスが境界内にあるかどうかチェックします。

CHKA は、APSR 条件コードフラグを変更しません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.5.2 HB, HBL

Handler Branch branches to a specified handler.

Handler Branch with Link saves a return address to the LR, and then branches to a specified handler.

Encoding E1 ThumbEE

```
HB{L}<c> #<HandlerID>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	1	L	handler							

```
generate_link = (L == '1');  handler_offset = ZeroExtend(handler:'00000', 32);
```

Assembler syntax

`HB<c><q>  #<HandlerID>`	Encoded as L = 0
`HBL<c><q>  #<HandlerID>`	Encoded as L = 1

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<HandlerID>` The index number of the handler to be called, in the range 0-255.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if generate_link then
        next_instr_addr = PC - 2;
        LR = next_instr_addr<31:1> : '1';
    BranchWritePC(TEEHBR + handler_offset);
```

Exceptions

None.

Usage

`HB{L}` makes a large number of handlers available.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.5.2 HB、HBL

HB（ハンドラ分岐）は、指定されたハンドラに分岐します。

HBL（リンク付きハンドラ分岐）は、復帰アドレスを LR に保存し、指定されたハンドラに分岐します。

エンコード E1 ThumbEE

```
HB{L}<c> #<HandlerID>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	1	L	handler							

```
generate_link = (L == '1');  handler_offset = ZeroExtend(handler:'00000', 32);
```

アセンブラ構文

`HB<c><q>  #<HandlerID>`	L = 0 としてエンコードされる
`HBL<c><q>  #<HandlerID>`	L = 1 としてエンコードされる

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<HandlerID>` 呼び出されるハンドラのインデクス番号（0 ～ 255）。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if generate_link then
        next_instr_addr = PC - 2;
        LR = next_instr_addr<31:1> : '1';
    BranchWritePC(TEEHBR + handler_offset);
```

例外

なし

使用法

`HB{L}` によって、多数のハンドラが使用可能になります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.5.3 HBLP

HBLP (Handler Branch with Link and Parameter) saves a return address to the LR, and then branches to a specified handler. It passes a 5-bit parameter to the handler in R8.

Encoding E1 ThumbEE

```
HBLP<c> #<imm>, #<HandlerID>
```

15 14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
1 1 0 0	0 1	imm5	handler

```
imm32 = ZeroExtend(imm5, 32);  handler_offset = ZeroExtend(handler:'00000', 32);
```

Assembler syntax

```
HBLP<c><q>  #<imm>, #<HandlerID>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<imm>`	The parameter to pass to the handler, in the range 0-31.
`<HandlerID>`	The index number of the handler to be called, in the range 0-31.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[8] = imm32;
    next_instr_addr = PC - 2;
    LR = next_instr_addr<31:1> : '1';
    BranchWritePC(TEEHBR + handler_offset);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.5.3 HBLP

HBLP（リンク / パラメータ付きハンドラ分岐）は、復帰アドレスを LR に保存し、指定されたハンドラに分岐します。5 ビットのパラメータを、R8 のハンドラに渡します。

エンコード E1 ThumbEE

```
HBLP<c> #<imm>, #<HandlerID>
```

15 14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
1 1 0 0	0 1	imm5	handler

```
imm32 = ZeroExtend(imm5, 32);  handler_offset = ZeroExtend(handler:'00000', 32);
```

アセンブラ構文

```
HBLP<c><q>  #<imm>, #<HandlerID>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<imm>`	ハンドラに渡されるパラメータ（0 ～ 31）。
`<HandlerID>`	呼び出されるハンドラのインデクス番号（0 ～ 31）。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[8] = imm32;
    next_instr_addr = PC - 2;
    LR = next_instr_addr<31:1> : '1';
    BranchWritePC(TEEHBR + handler_offset);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.5.4 HBP

HBP (Handler Branch with Parameter) causes a branch to a specified handler. It passes a 3-bit parameter to the handler in R8.

Encoding E1 ThumbEE

```
HBP<c> #<imm>, #<HandlerID>
```

15 14 13 12	11 10 9 8	7 6 5	4 3 2 1 0
1 1 0 0	0 0 0 0	imm3	handler

```
imm32 = ZeroExtend(imm3, 32);  handler_offset = ZeroExtend(handler:'00000', 32);
```

Assembler syntax

```
HBP<c><q>  #<imm>, #<HandlerID>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A8-7.
`<imm>`	The parameter to pass to the handler, in the range 0-7.
`<HandlerID>`	The index number of the handler to be called, in the range 0-31.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[8] = imm32;
    BranchWritePC(TEEHBR + handler_offset);
```

Exceptions

None.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.5.4 HBP

HBP（パラメータ付きハンドラ分岐）は、指定されたハンドラへの分岐を引き起こします。3 ビットのパラメータを、R8 のハンドラに渡します。

エンコード E1 ThumbEE

```
HBP<c> #<imm>, #<HandlerID>
```

15 14 13 12	11 10 9 8	7 6 5	4 3 2 1 0
1 1 0 0	0 0 0 0	imm3	handler

```
imm32 = ZeroExtend(imm3, 32);  handler_offset = ZeroExtend(handler:'00000', 32);
```

アセンブラ構文

```
HBP<c><q>  #<imm>, #<HandlerID>
```

各項目の説明については以下を参照して下さい。

`<c><q>`	*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。
`<imm>`	ハンドラに渡されるパラメータ（0 ～ 7）。
`<HandlerID>`	呼び出されるハンドラのインデクス番号（0 ～ 31）。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[8] = imm32;
    BranchWritePC(TEEHBR + handler_offset);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.5.5 LDR (immediate)

Load Register (immediate) provides 16-bit instructions to load words using:

- R9 as base register, with a positive offset of up to 63 words, for loading from a frame
- R10 as base register, with a positive offset of up to 31 words, for loading from a literal pool
- R0-R7 as base register, with a negative offset of up to 7 words, for array operations.

Encoding E1 ThumbEE

```
LDR<c> <Rt>,[R9{, #<imm>}]
```

15 14 13 12	11 10 9	8 7 6 5 4 3	2 1 0
1 1 0 0	1 1 0	imm6	Rt

```
t = UInt(Rt);  n = 9;  imm32 = ZeroExtend(imm6:'00', 32);  add = TRUE;
```

Encoding E2 ThumbEE

```
LDR<c> <Rt>,[R10{, #<imm>}]
```

15 14 13 12	11 10 9 8	7 6 5 4 3	2 1 0
1 1 0 0	1 0 1 1	imm5	Rt

```
t = UInt(Rt);  n = 10;  imm32 = ZeroExtend(imm5:'00', 32);  add = TRUE;
```

Encoding E3 ThumbEE

```
LDR<c> <Rt>,[<Rn>{, #-<imm>}]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
1 1 0 0	1 0 0	imm3	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm3:'00', 32);  add = FALSE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.5.5 LDR（イミディエート）

LDR（レジスタロード（イミディエート））は、次のベースレジスタを使用してワードをロードする 16 ビット命令です。

- フレームからロードする場合、R9（正のオフセットが最大 63 ワード）
- リテラルプールからロードする場合、R10（正のオフセットが最大 31 ワード）
- 配列操作の場合、R0 ～ R7（負のオフセットが最大 7 ワード）

エンコード E1 ThumbEE

```
LDR<c> <Rt>,[R9{, #<imm>}]
```

15 14 13 12	11 10 9	8 7 6 5 4 3	2 1 0
1 1 0 0	1 1 0	imm6	Rt

```
t = UInt(Rt);  n = 9;  imm32 = ZeroExtend(imm6:'00', 32);  add = TRUE;
```

エンコード E2 ThumbEE

```
LDR<c> <Rt>,[R10{, #<imm>}]
```

15 14 13 12	11 10 9 8	7 6 5 4 3	2 1 0
1 1 0 0	1 0 1 1	imm5	Rt

```
t = UInt(Rt);  n = 10;  imm32 = ZeroExtend(imm5:'00', 32);  add = TRUE;
```

エンコード E3 ThumbEE

```
LDR<c> <Rt>,[<Rn>{, #-<imm>}]
```

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
1 1 0 0	1 0 0	imm3	Rn	Rt

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm3:'00', 32);  add = FALSE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
LDR<c><q>  <Rt>, [<Rn>{, #<imm>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination register.

`<Rn>` The base register. This register is:
- R9 for encoding E1
- R10 for encoding E2
- any of R0-R7 for encoding E3.

`<imm>` The immediate offset used to form the address. Values are multiples of 4 in the range:

`0-252`	encoding E1
`0-124`	encoding E2
`-28-0`	encoding E3.

`<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    R[t] = MemU[address,4];
```

Exceptions and checks

Data Abort, NullCheck.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
LDR<c><q>  <Rt>, [<Rn>{, #<imm>}]
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションレジスタ

`<Rn>` ベースレジスタ。このレジスタは次のとおりです。
- R9（エンコード E1 の場合）
- R10（エンコード E2 の場合）
- R0 ～ R7 のいずれか（エンコード E3 の場合）

`<imm>` アドレスを形成するために使用されるイミディエートオフセット。値は 4 の倍数で、範囲は次のとおりです。

`0-252`	エンコード E1
`0-124`	エンコード E2
`-28-0`	エンコード E3

`<imm>` は省略可能で、その場合オフセットは 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(n);
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    R[t] = MemU[address,4];
```

例外およびチェック

データアボート、NULL チェック

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A9.5.6 STR (immediate)

Store Register (immediate) provides a 16-bit word store instruction using R9 as base register, with a positive offset of up to 63 words, for storing to a frame.

Encoding E1 ThumbEE

```
STR<c> <Rt>, [R9, #<imm>]
```

15 14 13 12	11 10 9	8 7 6 5 4 3	2 1 0
1 1 0 0	1 1 1	imm6	Rt

```
t = UInt(Rt);  imm32 = ZeroExtend(imm6:'00', 32);
```

Assembler syntax

```
STR<c><q>  <Rt>, [R9, #<imm>]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The source register.

`<imm>` The immediate offset applied to the value of R9 to form the address. Values are multiples of 4 in the range 0-252.

`<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(9);
    address = R[9] + imm32;
    MemU[address,4] = R[t];
```

Exceptions and checks

Data Abort, NullCheck.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A9.5.6 STR（イミディエート）

STR（レジスタストア（イミディエート））は、フレームにストアする場合に、ベースレジスタとして R9（正のオフセットが最大 63 ワード）を使用する 16 ビットワードストア命令です。

エンコード E1 ThumbEE

```
STR<c> <Rt>, [R9, #<imm>]
```

15 14 13 12	11 10 9	8 7 6 5 4 3	2 1 0
1 1 0 0	1 1 1	imm6	Rt

```
t = UInt(Rt);  imm32 = ZeroExtend(imm6:'00', 32);
```

アセンブラ構文

```
STR<c><q>  <Rt>, [R9, #<imm>]
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

`<Rt>` ソースレジスタ

`<imm>` アドレス生成のために R9 の値に適用されるイミディエートオフセット。値の範囲は、4 の倍数で 0 ～ 252 です。

`<imm>` は省略可能で、その場合オフセットは 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();  NullCheckIfThumbEE(9);
    address = R[9] + imm32;
    MemU[address,4] = R[t];
```

例外およびチェック

データアボート、NULL チェック

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Part B

System Level Architecture

抄訳

パート B

システムレベルのアーキテクチャ

Chapter B1
The System Level Programmers' Model

This chapter provides a system-level view of the programmers' model. It contains the following sections:

- *About the system level programmers' model* on page B1-2
- *System level concepts and terminology* on page B1-3
- *ARM processor modes and core registers* on page B1-6
- *Instruction set states* on page B1-23
- *The Security Extensions* on page B1-25
- *Exceptions* on page B1-30
- *Coprocessors and system control* on page B1-62
- *Advanced SIMD and floating-point support* on page B1-64
- *Execution environment support* on page B1-73.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 B1 章
システムレベルのプログラマモデル

本章では、システムレベルから見たプログラマモデルについて説明します。本章は次のセクションから構成されています。

- *システムレベルのプログラマモデルについて* P. B1-2
- *システムレベルの概念と用語* P. B1-3
- *ARM プロセッサモードとコアレジスタ* P. B1-6
- *命令セット状態* P. B1-23
- *セキュリティ拡張機能* P. B1-25
- *例外* P. B1-30
- *コプロセッサとシステム制御* P. B1-62
- *アドバンスト SIMD と浮動小数点のサポート* P. B1-64
- *実行環境のサポート* P. B1-73

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.1 About the system level programmers' model

An application programmer has only a restricted view of the system. The system level programmers' model supports this application level view of the system, and includes features required for an operating system (OS) to provide the programming environment seen by an application.

The system level programmers' model includes all of the system features required to support operating systems and to handle hardware events.

System level concepts and terminology on page B1-3 gives a system level introduction to the basic concepts of the ARM architecture, and the terminology used to describe the architecture. The rest of this chapter describes the system level programmers' model.

The other chapters in this part describe:

- The memory system architectures:
  - Chapter B2 *Common Memory System Architecture Features* describes common features of the memory system architectures
  - Chapter B3 *Virtual Memory System Architecture (VMSA)* describes the *Virtual Memory System Architecture* (VMSA) used in the ARMv7-A profile
  - Chapter B4 *Protected Memory System Architecture (PMSA)* describes the *Protected Memory System Architecture* (PMSA) used in the ARMv7-R profile.
- The CPUID mechanism, that enables an OS to determine the capabilities of the processor it is running on. See Chapter B5 *The CPUID Identification Scheme*.
- The instructions that provide system-level functionality, such as returning from an exception. See Chapter B6 *System Instructions*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B1.1 システムレベルのプログラマモデルについて

アプリケーションプログラマは、システムの一部を把握しているに過ぎません。システムレベルのプログラマモデルは、このようなアプリケーションレベルで見られるシステムを支えるものです。このモデルには、アプリケーションから見えるプログラミング環境を提供するためにオペレーティングシステム (OS) に必要な機能が備わっています。

システムレベルのプログラマモデルには、オペレーティングシステムをサポートし、ハードウェアイベントを処理するために必要なすべてのシステム機能が含まれています。

システムレベルの概念と用語 P. B1-3 では、ARM アーキテクチャの基本概念のシステムレベルの概要と、アーキテクチャの説明で使用される用語について説明します。本章の残りの部分では、システムレベルのプログラマモデルについて説明します。

他の章では、次の内容について説明します。

- メモリシステムのアーキテクチャ
  - 第 B2 章 *「一般的なメモリシステム アーキテクチャの機能」* では、メモリシステムのアーキテクチャの一般的な機能について説明します。
  - 第 B3 章 *「仮想メモリシステム アーキテクチャ (VMSA)」* では、ARMv7-A プロファイルで使用される *仮想メモリシステム アーキテクチャ* (VMSA) について説明します。
  - 第 B4 章 *「保護メモリシステム アーキテクチャ (PMSA)」* では、ARMv7-R プロファイルで使用される *保護メモリシステム アーキテクチャ* (PMSA) について説明します。
- OS から、自分が実行されているプロセッサの機能の特定を可能にする CPUID 機構。第 B5 章 *「CPUID 識別方式」* を参照して下さい。
- 例外からの復帰など、システムレベルの機能を提供する命令。第 B6 章 *「システム命令」* を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.2 System level concepts and terminology

A number of concepts are critical to understanding the system level architecture support. These are introduced in the following sections:

- *Privilege, mode, and state*
- *Exceptions* on page B1-4

B1.2.1 Privilege, mode, and state

Privilege, mode, and state are key concepts in the ARM architecture.

Privilege

Software can execute as privileged or unprivileged:

- Unprivileged execution limits or excludes access to some resources in the current security state.
- Privileged execution gives access to all resources in the current security state.

Mode

The ARM architecture provides a set of modes that support normal software execution and handle exceptions. The current mode determines the set of registers that are available and the privilege of the executing software. For more information, see *ARM processor modes and core registers* on page B1-6.

State

In the ARM architecture, *state* is used to describe the following distinct concepts:

Instruction set state

ARMv7 provides four instruction set states. The instruction set state determines the instruction set that is being executed, and is one of ARM state, Thumb state, Jazelle state, or ThumbEE state. *ISETSTATE* on page A2-15 gives more information about these states.

Execution state

The execution state consists of the instruction set state and some control bits that modify how the instruction stream is decoded. For details, see *Execution state registers* on page A2-15 and *Program Status Registers (PSRs)* on page B1-14.

Security state In the ARM architecture, the number of security states depends on whether the Security Extensions are implemented:

- When the Security Extensions are implemented, the ARM architecture provides two security states, Secure state and Non-secure state. Each security state has its own system registers and memory address space.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.2 システムレベルの概念と用語

システムレベルのアーキテクチャ サポートを理解する上で不可欠な概念がいくつかあります。これらの概念については、次のセクションで紹介します。

- *特権、モード、状態*
- *例外* P. B1-4

B1.2.1 特権、モード、状態

特権、モード、状態は、ARM アーキテクチャの重要な概念です。

特権

ソフトウェアは、特権または非特権として実行することができます。

- 非特権実行では、現在のセキュリティ状態での一部のリソースへのアクセスが制限または除外されます。
- 特権実行では、現在のセキュリティ状態でのすべてのリソースへのアクセスが可能です。

モード

ARM アーキテクチャでは、正常なソフトウェア実行をサポートし、例外を処理する一連のモードが提供されます。現在のモードによって、利用可能なレジスタのセットとソフトウェア実行の特権が決定されます。詳細については、*ARM プロセッサモードとコアレジスタ* P. B1-6 を参照して下さい。

状態

ARM アーキテクチャでは、次のような異なる概念を説明するために*状態*という用語が使用されます。

命令セット状態

ARMv7 では、4 つの命令セット状態が提供されます。命令セット状態によって、実行する命令セットが決定されます。命令セット状態は、ARM 状態、Thumb 状態、Jazelle 状態、または ThumbEE 状態のいずれかです。*ISETSTATE* P. A2-15 で、これらの状態について詳しく説明します。

実行状態 実行状態は、命令セット状態と、命令ストリームのデコード方法を変更するいくつかの制御ビットで構成されます。詳細については、*実行状態レジスタ* P. A2-15 および*プログラムステータス レジスタ (PSR)* P. B1-14 を参照して下さい。

セキュリティ状態

ARM アーキテクチャでは、セキュリティ拡張機能が実装されているかどうかによってセキュリティ状態の数が異なります。

- セキュリティ拡張機能が実装されている場合は、セキュア状態と非セキュア状態の 2 つのセキュリティ状態が提供されます。セキュリティ状態ごとに、システムレジスタとメモリアドレス空間が割り当てられます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The security state is largely independent of the processor mode. The only exception to this independence of security state and processor mode is Monitor mode. Monitor mode exists only in the Secure state, and supports transitions between Secure and Non-secure state.

Some system control resources are only accessible from the Secure state.

For more information, see *The Security Extensions* on page B1-25.

Note

In some documentation, the Secure state is described as the *Secure world*, and the Non-secure state is described as the *Non-secure world*.

- When the Security Extensions are not implemented, the ARM architecture provides only a single security state.

Debug state Debug state refers to the processor being halted for debug purposes, because a debug event has occurred when the processor is configured to Halting debug-mode. See *Invasive debug* on page C1-3.

When the processor is not in Debug state it is in Non-debug state.

Except where explicitly stated otherwise, parts A and B of this manual describe processor behavior and instruction execution in Non-debug state. Chapter C5 *Debug State* describes the differences in Debug state.

B1.2.2 Exceptions

An exception is a condition that changes the normal flow of control in a program. The change of flow switches execution to an exception handler, and the state of the system at the point where the exception occurred is presented to the exception handler. A key component of the state presented to the handler is the return address, that indicates the point in the instruction stream where the exception was taken.

The ARM architecture provides a number of different exceptions as described in *Exceptions* on page B1-30.

Terminology for describing exceptions

In this manual, a number of terms have specific meanings when used to describe exceptions:

- An exception is *generated* in one of the following ways:
  - — Directly as a result of the execution or attempted execution of the instruction stream. For example, an exception is generated as a result of an UNDEFINED instruction.
  - — Less directly, as a result of something in the state of the system. For example, an exception is generated as a result of an interrupt signaled by a peripheral.
- An exception is *taken* by a processor at the point where it causes a change to the normal flow of control in the program.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

セキュリティ状態は、プロセッサモードの影響をほとんど受けません。セキュリティ状態とプロセッサモードに依存関係がある唯一の例外はモニタモードです。モニタモードはセキュア状態でのみ存在し、セキュア状態と非セキュア状態との間の遷移をサポートします。

システム制御リソースの中には、セキュア状態からしかアクセスできないものがあります。

詳細については、*セキュリティ拡張機能* P. B1-25 を参照して下さい。

注

説明書によっては、セキュア状態が*セキュアワールド*、非セキュア状態が*非セキュアワールド*と記述されている場合があります。

- セキュリティ拡張機能が実装されていない場合は、単一のセキュリティ状態しか提供されません。

デバッグ状態 デバッグ状態とは、デバッグのためにプロセッサが停止中であることを意味します。この状態は、プロセッサがホールト デバッグモードに構成されているときに、デバッグイベントが発生したことで遷移します。詳細については、*侵襲性デバッグ* P. C1-3 を参照して下さい。

プロセッサがデバッグ状態でない場合は、非デバッグ状態です。

明記されている場合を除いて、本書のパート A とパート B には、非デバッグ状態でのプロセッサの動作と命令実行について記載されています。デバッグ状態との違いについては、第 C5 章 *「デバッグ状態」*を参照して下さい。

B1.2.2 例外

例外は、プログラムの通常のフロー制御を変更する条件です。フローの変更によって、実行が例外ハンドラに切り替えられ、例外が発生した時点のシステムの状態が例外ハンドラに渡されます。ハンドラに渡される状態の重要な要素の 1 つが復帰アドレスで、これは命令ストリーム内で例外が検出された場所を示します。

ARM アーキテクチャでは、*例外* P. B1-30 に記載されているような数多くの例外が提供されます。

例外に関する用語

本書では、例外の説明で使用されるさまざまな用語に特定の意味合いを持たせています。

- 例外は、次のいずれかの方法で*生成*されます。
  - – 命令ストリームの実行または試行の結果として直接的に生成される。例えば、未定義命令の結果として例外が生成されます。
  - – 何らかのシステムの状態の結果としてより間接的に生成される。例えば、ペリフェラルから通知された割り込みの結果として例外が生成されます。
- 例外は、プログラム内の通常の制御フローが変化した時点でプロセッサによって*取得*されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- An exception is described as *synchronous* if both of the following apply:
  - — the exception is generated as a result of direct execution or attempted execution of the instruction stream
  - — the return address presented to the exception handler is guaranteed to indicate the instruction that caused the exception.
- An exception is described as *asynchronous* if either of the following applies:
  - — the exception is not generated as a result of direct execution or attempted execution of the instruction stream
  - — the return address presented to the exception handler is not guaranteed to indicate the instruction that caused the exception.

Asynchronous exceptions are of two types:

- a *precise asynchronous exception* guarantees that the state presented to the exception handler is consistent with the state at an identifiable instruction boundary in the execution stream from which the exception was taken.
- an *imprecise asynchronous exception* is one where the state presented to the exception handler is not guaranteed to be consistent with any point in the execution stream from which the exception was taken.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- 次の両方に該当する場合、その例外を*同期*と呼びます。
  - – その例外が、命令ストリームの直接実行または実行の試みの結果として生成される。
  - – 例外ハンドラに渡された復帰アドレスが、例外を引き起こした命令を示していることが保証される。
- 次のいずれかに該当する場合、その例外を*非同期*と呼びます。
  - – その例外が、命令ストリームの直接実行または実行の試みの結果として生成されない。
  - – 例外ハンドラに渡された復帰アドレスが、例外を引き起こした命令を示していることが保証されない。

非同期例外には、次の 2 種類があります。

- *正確な非同期例外*では、例外ハンドラに渡された状態と、例外が取得された実行ストリーム内の特定可能な命令境界での状態が一致することが保証されます。
- *不正確な非同期例外*は、例外ハンドラに渡された状態と、例外が取得された実行ストリーム内の場所での状態が一致することが保証されない例外です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.3 ARM processor modes and core registers

The following sections describe the ARM processor modes and the core registers:

- *ARM processor modes*
- *ARM core registers* on page B1-9
- *Program Status Registers (PSRs)* on page B1-14.

B1.3.1 ARM processor modes

The ARM architecture defines eight modes, shown in Table B1-1:

Table B1-1 ARM processor modes

Processor mode[a]		Mode encoding[b]	Privilege	Description
User	usr	10000	Unprivileged	Suitable for most application code.
FIQ	fiq	10001	Privileged	Entered as a result of a fast interrupt.[c]
IRQ	irq	10010	Privileged	Entered as a result of a normal interrupt.[c]
Supervisor	svc	10011	Privileged	Suitable for running most kernel code. Entered on Reset, and on execution of a Supervisor Call (SVC) instruction.
Monitor[d]	mon	10110	Privileged	A Secure mode that enables change between Secure and Non-secure states, and can also be used to handle any of FIQs, IRQs and external aborts.[c] Entered on execution of a Secure Monitor Call (SMC) instruction.
Abort	abt	10111	Privileged	Entered as a result of a Data Abort exception or Prefetch Abort exception.[c]
Undefined	und	11011	Privileged	Entered as a result of an instruction-related error.
System	sys	11111	Privileged	Suitable for application code that requires privileged access.

a. Processor mode names and abbreviations.
b. CPSR.M. All other values are reserved. When the Security Extensions are not implemented the Monitor mode encoding, 0b10110, is reserved.
c. Bits in the Secure Configuration Register can be set so that one or more of FIQs, IRQs and external aborts are handled in Monitor mode, see *c1, Secure Configuration Register (SCR)* on page B3-106.
d. Only supported when the Security Extensions are implemented.

Mode changes can be made under software control, or can be caused by an external or internal exception.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B1.3 ARM プロセッサモードとコアレジスタ

ARM プロセッサモードとコアレジスタについては、次のセクションを参照して下さい。

- *ARM プロセッサモード*
- *ARM コアレジスタ* P. B1-9
- *プログラムステータス レジスタ (PSR)* P. B1-14

B1.3.1 ARM プロセッサモード

ARM アーキテクチャでは、表 B1-1 に示す 8 つのモードが定義されています。

表 B1-1 ARM プロセッサモード

プロセッサモード[a]		モードのエンコード[b]	特権	説明
ユーザ	usr	10000	非特権	ほとんどのアプリケーションコードに適しています。
FIQ	fiq	10001	特権	高速割り込みの結果として遷移します。[c]
IRQ	irq	10010	特権	通常割り込みの結果として遷移します。[c]
スーパバイザ	svc	10011	特権	ほとんどのカーネルコードの実行に適しています。 リセットと、スーパバイザコール (SVC) 命令の実行により遷移します。
モニタ[d]	mon	10110	特権	セキュア状態と非セキュア状態の間の遷移を可能にし、任意の FIQ、IRQ、および外部アボートの処理に使用可能なセキュアモードです。[c] セキュア モニタコール (SMC) 命令の実行により遷移します。
アボート	abt	10111	特権	データアボート例外またはプリフェッチアボート例外の結果として遷移します。[c]
未定義	und	11011	特権	命令関連のエラーの結果として遷移します。
システム	sys	11111	特権	特権アクセスが必要なアプリケーションコードに適しています。

a. プロセッサモードの名前と省略形
b. CPSR.M。他の値はすべて予約済みです。セキュリティ拡張機能が実装されていない場合は、モニタモードのエンコードである 0b10110 は予約済みです。
c. セキュア コンフィギュレーションレジスタ内のビットは、1 つ以上の FIQ、IRQ、および外部アボートがモニタモードで処理されるようにセットすることができます。*c1、セキュア構成レジスタ (SCR)* P. B3-106 を参照して下さい。
d. セキュリティ拡張機能が実装されている場合にのみサポートされます。

モード変更は、ソフトウェア制御下で行うことも、外部または内部の例外によって引き起こすこともできます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Notes on the ARM processor modes

User mode User mode enables the operating system to restrict the use of system resources. Application programs normally execute in User mode. In User mode, the program being executed:

- cannot access protected system resources
- cannot change mode except by causing an exception, see *Exceptions* on page B1-30.

Privileged modes

The modes other than User mode are known as *privileged modes*. In their security state they have full access to system resources and can change mode freely.

Exception modes

The exception modes are:

- FIQ mode
- IRQ mode
- Supervisor mode
- Abort mode
- Undefined mode
- Monitor mode.

Each of these modes normally handles the corresponding exceptions, as shown in Table B1-1 on page B1-6.

Each exception mode has some banked registers to avoid corrupting the registers of the mode in use when the exception is taken, see *ARM core registers* on page B1-9.

System mode System mode has the same registers available as User mode, and is not entered by any exception.

System mode is intended for use by operating system tasks that must access system resources, but do not want to use the exception entry mechanism and the associated additional registers. Also, it is used when the operating system has to access the User mode registers.

Monitor mode

Monitor mode is only implemented as part of the Security Extensions, and is always in the Secure state, regardless of the value of the SCR.NS bit. For more information, see *The Security Extensions* on page B1-25.

Code running in Monitor mode has access to both the Secure and Non-secure copies of system registers. This means Monitor mode provides the normal method of changing between the Secure and Non-secure security states.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARM プロセッサモードに関する注意事項

ユーザモード ユーザモードでは、オペレーティングシステムがシステムリソースの使用を制限することができます。アプリケーションプログラムは、通常はユーザモードで動作します。ユーザモードで実行されるプログラムには、次のような制限があります。

- 保護されたシステムリソースにアクセスできません。
- 例外以外でモードを変更できません。*例外* P. B1-30 を参照して下さい。

特権モード ユーザモード以外のモードは*特権モード*と呼ばれます。特権モードのセキュリティ状態では、システムリソースにフルアクセスすることができ、自由にモードを変更できます。

例外モード 例外モードは次のとおりです。

- FIQ モード
- IRQ モード
- スーパバイザモード
- アボートモード
- 未定義モード
- モニタモード

P. B1-6 の表 B1-1 に示すように、これらのモードは通常、それぞれに対応する例外を処理します。

各例外モードでは、いくつかのバンクレジスタを使用して、例外が取得された時点で使用中だったモードのレジスタの破壊を回避することができます。*ARM コアレジスタ* P. B1-9 を参照して下さい。

システムモード

システムモードではユーザモードと同じレジスタを使用することができます。例外によってシステムモードに遷移することはありません。

システムモードは、システムリソースにアクセスする必要があり、例外の開始機構と関連する追加レジスタは使用しないオペレーティングシステム タスクで使用されるモードです。また、オペレーティングシステムでユーザモード レジスタにアクセスする必要があるときにも使用されます。

モニタモード モニタモードは、セキュリティ拡張機能の一部としてのみ実装され、SCR.NS ビットの値に関係なく、常にセキュア状態にあります。詳細については、*セキュリティ拡張機能* P. B1-25 を参照して下さい。

モニタモードで実行中のコードは、システムレジスタのセキュアコピーと非セキュアコピーの両方にアクセスすることができます。これは、モニタモードによって、セキュアと非セキュアのセキュリティ状態を切り替える標準的手段が提供されることを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Secure and Non-secure modes

In a processor that implements the Security Extensions, a mode description can be qualified as Secure or Non-secure, to indicate whether the processor is also in Secure state or Non-secure state. For example:

- if a processor is in a privileged mode and Secure state, it is in a *Secure privileged mode*
- if a processor is in User mode and Non-secure state, it is in *Non-secure User mode*.

Pseudocode details of mode operations

The `BadMode()` function tests whether a 5-bit mode number corresponds to one of the permitted modes:

```
// BadMode()
// =========

boolean BadMode(bits(5) mode)
    case mode of
        when '10000'  result = FALSE;                // User mode
        when '10001'  result = FALSE;                // FIQ mode
        when '10010'  result = FALSE;                // IRQ mode
        when '10011'  result = FALSE;                // Supervisor mode
        when '10110'  result = !HaveSecurityExt();   // Monitor mode
        when '10111'  result = FALSE;                // Abort mode
        when '11011'  result = FALSE;                // Undefined mode
        when '11111'  result = FALSE;                // System mode
        otherwise     result = TRUE;
    return result;
```

The following pseudocode functions provide information about the current mode:

```
// CurrentModeIsPrivileged()
// =========================

boolean CurrentModeIsPrivileged()
    if BadMode(CPSR.M) then UNPREDICTABLE;
    if CPSR.M == '10000' then return FALSE;  // User mode
    return TRUE;                             // Other modes

// CurrentModeIsUserOrSystem()
// ===========================

boolean CurrentModeIsUserOrSystem()
    if BadMode(CPSR.M) then UNPREDICTABLE;
    if CPSR.M == '10000' then return TRUE;  // User mode
    if CPSR.M == '11111' then return TRUE;  // System mode
    return FALSE;                           // Other modes
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

セキュアモードと非セキュアモード

セキュリティ拡張機能が実装されたプロセッサでは、プロセッサがセキュア状態と非セキュア状態のどちらにあるかを示すために、モードの記述をセキュアまたは非セキュアで修飾することができます。次に例を示します。

- プロセッサが特権モードでセキュア状態の場合、*セキュア特権モード*です。
- プロセッサがユーザモードで非セキュア状態の場合、*非セキュア ユーザモード*です。

モード動作の擬似コードの詳細

`BadMode()` 関数は、5 ビットのモード値が、許可されたモードのいずれかと一致するかどうかをテストします。

```
// BadMode()
// =========

boolean BadMode(bits(5) mode)
    case mode of
        when '10000'  result = FALSE;                // User mode
        when '10001'  result = FALSE;                // FIQ mode
        when '10010'  result = FALSE;                // IRQ mode
        when '10011'  result = FALSE;                // Supervisor mode
        when '10110'  result = !HaveSecurityExt();   // Monitor mode
        when '10111'  result = FALSE;                // Abort mode
        when '11011'  result = FALSE;                // Undefined mode
        when '11111'  result = FALSE;                // System mode
        otherwise     result = TRUE;
    return result;
```

次の擬似コード関数は、現在のモードに関する情報を提供します。

```
// CurrentModeIsPrivileged()
// =========================

boolean CurrentModeIsPrivileged()
    if BadMode(CPSR.M) then UNPREDICTABLE;
    if CPSR.M == '10000' then return FALSE;  // User mode
    return TRUE;                             // Other modes

// CurrentModeIsUserOrSystem()
// ===========================

boolean CurrentModeIsUserOrSystem()
    if BadMode(CPSR.M) then UNPREDICTABLE;
    if CPSR.M == '10000' then return TRUE;  // User mode
    if CPSR.M == '11111' then return TRUE;  // System mode
    return FALSE;                           // Other modes
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.3.2 ARM core registers

ARM core registers on page A2-11 describes the application level view of the ARM register file. This view provides 16 ARM core registers, R0 to R15, that include the *Stack Pointer* (SP), *Link Register* (LR), and *Program Counter* (PC). These registers are selected from a total set of either 31 or 33 registers, depending on whether or not the Security Extensions are implemented. The current execution mode determines the selected set of registers, as shown in Figure B1-1. This shows that the arrangement of the registers provides duplicate copies of some registers, with the current register selected by the execution mode. This arrangement is described as *banking* of the registers, and the duplicated copies of registers are referred to as *banked* registers:

System level views
Privileged modes
Exception modes

Application level view	User mode	System mode	Supervisor mode	Monitor mode ‡	Abort mode	Undefined mode	IRQ mode	FIQ mode
R0	R0_usr							
R1	R1_usr							
R2	R2_usr							
R3	R3_usr							
R4	R4_usr							
R5	R5_usr							
R6	R6_usr							
R7	R7_usr							
R8	R8_usr							R8_fiq
R9	R9_usr							R9_fiq
R10	R10_usr							R10_fiq
R11	R11_usr							R11_fiq
R12	R12_usr							R12_fiq
SP	SP_usr		SP_svc	SP_mon ‡	SP_abt	SP_und	SP_irq	SP_fiq
LR	LR_usr		LR_svc	LR_mon ‡	LR_abt	LR_und	LR_irq	LR_fiq
PC	PC							
APSR	CPSR							
			SPSR_svc	SPSR_mon ‡	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

‡ Monitor mode, and the associated banked registers, are implemented only as part of the Security Extensions

Figure B1-1 Organization of general-purpose registers and Program Status Registers

Figure B1-1 includes the views of the *Current Program Status Register* (CPSR) and of the banked *Saved Program Status Register* (SPSR), see *Program Status Registers (PSRs)* on page B1-14.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.3.2 ARM コアレジスタ

ARM コアレジスタ P. A2-11 は、アプリケーションレベルから見た ARM レジスタファイルの説明です。このビューでは、*スタックポインタ*(SP)、*リンクレジスタ*(LR)、*プログラムカウンタ*(PC) を含む、16 個の ARM コアレジスタの R0 ～ R15 が提供されます。これらのレジスタは、セキュリティ拡張機能が実装されているかどうかに基づいて、全部で 31 個または 33 個のレジスタセットから選択されます。図 B1-1 に示すように、現在の実行モードによって、レジスタの選択セットが決定されます。これは、レジスタの割り振りによって、実行モードにより選択される現在のレジスタと一部のレジスタのコピーが重複することを示しています。この割り振りは、レジスタの*バンク処理*と呼ばれ、重複したレジスタのコピーは*バンクレジスタ*と呼ばれます。

システムレベルのビュー
特権モード
例外モード

アプリケーションレベルのビュー	ユーザモード	システムモード	スーパバイザモード	モニタモード ‡	アボートモード	未定義モード	IRQ モード	FIQ モード
R0	R0_usr							
R1	R1_usr							
R2	R2_usr							
R3	R3_usr							
R4	R4_usr							
R5	R5_usr							
R6	R6_usr							
R7	R7_usr							
R8	R8_usr							R8_fiq
R9	R9_usr							R9_fiq
R10	R10_usr							R10_fiq
R11	R11_usr							R11_fiq
R12	R12_usr							R12_fiq
SP	SP_usr		SP_svc	SP_mon ‡	SP_abt	SP_und	SP_irq	SP_fiq
LR	LR_usr		LR_svc	LR_mon ‡	LR_abt	LR_und	LR_irq	LR_fiq
PC	PC							
APSR	CPSR							
			SPSR_svc	SPSR_mon ‡	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

‡ モニタモード、および関連するバンクレジスタはセキュリティ拡張機能の一部としてのみ実装されます。

図 B1-1 汎用レジスタとプログラムステータス レジスタの編成

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

- System level register names, such as R0_usr, R8_usr, and R8_fiq, are used when it is necessary to identify a specific register. The Application level names refer to the registers for the current mode, and usually are sufficient to identify a register.
- In ARMv7, the Security Extensions can be implemented only as part of an ARMv7-A implementation.

Each of the exception modes selects a different copy of the banked SP and LR, because these registers have special functions on exception entry:

SP This enables the exception handler to use a different stack to the one in use when the exception occurred. For example, it can use a stack in privileged memory rather than one in unprivileged memory.

LR The exception return address is placed in the banked LR of the exception mode. This means the use of the LR by the application is not corrupted. The address placed in the banked LR is at an exception-dependent offset from the next instruction to be executed in the code in which the exception occurred. This address enables the exception handler to return to that code, so the processor can resume execution of the code. Table B1-4 on page B1-34 shows the LR value saved on entry to each of the exception modes.

In addition:

- FIQ mode provides its own mappings for the general-purpose registers R8 to R12. These enable very fast processing of interrupts that are simple enough to be processed using only registers R8 to R12, SP, LR, and PC, without affecting the corresponding registers of the mode in which the interrupt was taken.
- In an exception mode the processor can access the SPSR for that mode. There is no SPSR for User mode and System mode.

In all ARMv7-A and ARMv7-R implementations:

- Every mode except User mode is *privileged*.
- User mode and System mode share the same register file. The only difference between System and User modes is that System mode runs with privileged access.

For more information about the application level view of the SP, LR, and the Program Counter (PC), and the alternative descriptions of them as R13, R14 and R15, see *ARM core registers* on page A2-11.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

図 B1-1 には、*カレントプログラム ステータスレジスタ*(CPSR) と*保存プログラム ステータス レジスタ*(SPSR) のビューが含まれています。*プログラムステータス レジスタ (PSR)* P. B1-14 を参照して下さい。

注

- 特定のレジスタを識別する必要がある場合は、R0_usr、R8_usr、R8_fiq などのシステムレベルのレジスタ名が使用されます。アプリケーションレベルの名前は、現在のモードのレジスタを参照しており、通常はレジスタを識別するためにこれで十分です。
- ARMv7 では、ARMv7-A 実装の一部としてのみセキュリティ拡張機能を実装できます。

例外モードごとに異なるバンク SP および LR のコピーが選択されます。これは、これらのレジスタが例外の開始に対して特別な機能を持っているためです。

SP これによって、例外ハンドラは、例外が発生したときに使用中のスタックとは別のスタックを使用することができます。例えば、非特権メモリではなく、特権メモリのスタックを使用することができます。

LR 例外からの復帰アドレスは、例外モードのバンク LR に置かれます。これは、アプリケーションによる LR の使用が影響を受けないことを意味します。バンク LR に置かれるアドレスは、例外が発生したコード内で次に実行される命令のアドレスに、例外ごとに異なるオフセットを加えた値です。このアドレスによって、例外ハンドラはそのコードに戻り、プロセッサはコードの実行を再開することができます。各例外モードに遷移するときに保存される LR の値を、P. B1-34 の表 B1-4 に示します。

また、次の規則が適用されます。

- FIQ モードでは、汎用レジスタの R8 ～ R12 に対する独自のマッピングが提供されます。これによって、レジスタの R8 ～ R12、SP、LR、および PC だけを使用して、割り込みが取得されたモードの対応するレジスタに影響を与えることなく、簡単に処理できるため、割り込み処理の高速化が可能になります。
- 例外モードでは、プロセッサからそのモードの SPSR にアクセスすることができます。ユーザモードとシステムモード用の SPSR は存在しません。

すべての ARMv7-A 実装と ARMv7-R 実装では、次の規則が適用されます。

- ユーザモード以外のすべてのモードが*特権モード*です。
- ユーザモードとシステムモードで同じレジスタファイルが共有されます。システムモードとユーザモードとの唯一の違いは、システムモードは特権アクセスで実行されることです。

アプリケーションレベルから見た SP、LR、プログラムカウンタ (PC) に関する詳細と、それらの R13、R14、R15 としての代替記述については、*ARM コアレジスタ* P. A2-11 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Writing to the PC

In ARMv7, instruction writes to the PC are handled as follows:

- Exception return instructions write both the PC and the CPSR. The value written to the CPSR determines the new instruction set state, and the value written to the PC determines the address that is branched to. For full details, including which instructions are exception return instructions and how incorrectly aligned PC values are handled, see *Exception return* on page B1-38.
- The following two 16-bit Thumb instruction encodings remain in Thumb state and branch to a value written to the PC:
  - — encoding T2 of *ADD (register)* on page A8-24
  - — encoding T1 of *MOV (register)* on page A8-196.

  The value written to the PC is forced to be halfword-aligned by ignoring its least significant bit, instead treating that bit as being 0.
- The following instructions remain in the same instruction set state and branch to a value written to the PC:
  - — `B`, `BL`, `CBNZ`, `CBZ`, `CHKA`, `HB`, `HBL`, `HBLP`, `HBP`, `TBB`, and `TBH`
  - — in ThumbEE state, load/store instructions that fail their null check.

  The definition of each of these instructions ensures that the value written to the PC is correctly aligned for the current instruction set state.
- The `BLX` (immediate) instruction switches between ARM and Thumb states and branches to a value written to the PC. Its definition ensures that the value written to the PC is correctly aligned for the new instruction set state.
- The following instructions write a value to the PC, treating that value as an interworking address with low-order bits that determine the new instruction set state and an address to branch to:
  - — `BLX` (register), `BX`, and `BXJ`
  - — `LDR`, and `LDRT` instructions with <Rt> equal to the PC
  - — `POP` and all forms of `LDM` except `LDM` (exception return), when the register list includes the PC
  - — in ARM state only, `ADC`, `ADD`, `ADR`, `AND`, `ASR` (immediate), `BIC`, `EOR`, `LSL` (immediate), `LSR` (immediate), `MOV`, `MVN`, `ORR`, `ROR` (immediate), `RRX`, `RSB`, `RSC`, `SBC`, and `SUB` instructions with <Rd> equal to the PC and without flag setting specified.

  For details of how an interworking address specifies the new instruction set state and instruction address, see *Pseudocode details of operations on ARM core registers* on page A2-12.

  Note

  — The `LDR`, `LDRT`, `POP`, and `LDM` instructions first have this behavior in ARMv5T.

  — The instructions listed as having this behavior in ARM state only first have this behavior in ARMv7.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

PC への書き込み

ARMv7 では、命令による PC への書き込みは次のように処理されます。

- 例外からの復帰命令は、PC と CPSR の両方に書き込みます。CPSR に書き込まれた値によって新しい命令セット状態が決定され、PC に書き込まれた値によって分岐先のアドレスが決定されます。どの命令が例外からの復帰命令に該当するか、および不正にアラインされた PC 値がどのように処理されるかについての詳細については、*例外からの復帰* P. B1-38 を参照して下さい。
- 次の 2 つの 16 ビット Thumb 命令のエンコードは、Thumb 状態を維持したまま、PC に書き込まれた値に分岐します。
  - – *ADD（レジスタ）* P. A8-24 のエンコード T2
  - – *MOV（レジスタ）* P. A8-196 のエンコード T1

  PC に書き込まれる値は、最下位ビットが 0 として処理されるのではなく、無視されることによって、ハーフワードアラインが強制されます。
- 次の命令は、同じ命令セット状態を維持したまま、PC に書き込まれた値に分岐します。
  - – `B`、`BL`、`CBNZ`、`CBZ`、`CHKA`、`HB`、`HBL`、`HBLP`、`HBP`、`TBB`、`TBH`
  - – ThumbEE 状態で、NULL チェックに失敗したロード / ストア命令

  これらの各命令の定義によって、PC に書き込まれた値がその時点の命令セット状態に応じて正しくアラインされることが保証されます。
- `BLX`（イミディエート）命令は、ARM 状態と Thumb 状態を切り替え、PC に書き込まれた値に分岐します。この命令の定義によって、PC に書き込まれた値が新しい命令セット状態に応じて正しくアラインされることが保証されます。
- 次の命令が PC に書き込む値は、新しい命令セット状態と分岐先アドレスを決定する下位ビットを使用したインターワーキングアドレスとして処理されます。
  - – `BLX`（レジスタ）、`BX`、`BXJ`
  - – <Rt> が PC と一致する `LDR` 命令と `LDRT` 命令
  - – `POP` と、`LDM`（例外からの復帰）以外の形式の `LDM`（レジスタリストに PC が含まれる場合）
  - – ARM 状態でのみ、<Rd> が PC と一致し、フラグが設定されていない `ADC` 命令、`ADD` 命令、`ADR` 命令、`AND` 命令、`ASR`（イミディエート）命令、`BIC` 命令、`EOR` 命令、`LSL`（イミディエート）命令、`LSR`（イミディエート）命令、`MOV` 命令、`MVN` 命令、`ORR` 命令、`ROR`（イミディエート）命令、`RRX` 命令、`RSB` 命令、`RSC` 命令、`SBC` 命令、`SUB` 命令

  インターワーキングアドレスによって新しい命令セット状態と命令アドレスがどのように指定されるかについては、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 を参照して下さい。

  注

  – `LDR`、`LDRT`、`POP`、`LDM` 命令がこの動作を行うのは ARMv5T およびそれ以降です。

  – ARM 状態でのみこの動作を行うものとして列挙されている命令は、ARMv7 からこの動作を行うようになりました。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In both cases, the behavior in earlier architecture versions is a branch that remains in the same instruction set state. For more information, see:

— *Interworking* on page AppxG-4, for ARMv6

— *Interworking* on page AppxH-5, for ARMv5 and ARMv4.

Pseudocode details of ARM core register operations

The following pseudocode gives access to the general-purpose registers:

```
// The names of the banked core registers.

enumeration RName {RName_0usr, RName_1usr, RName_2usr, RName_3usr, RName_4usr, RName_5usr,
                   RName_6usr, RName_7usr, RName_8usr, RName_8fiq, RName_9usr, RName_9fiq,
                   RName_10usr, RName_10fiq, RName_11usr, RName_11fiq, RName_12usr, RName_12fiq,
                   RName_SPusr, RName_SPfiq, RName_SPirq, RName_SPsvc,
                   RName_SPabt, RName_SPund, RName_SPmon,
                   RName_LRusr, RName_LRfiq, RName_LRirq, RName_LRsvc,
                   RName_LRabt, RName_LRund, RName_LRmon,
                   RName_PC};

// The physical array of banked core registers.
//
// _R[RName_PC] is defined to be the address of the current instruction. The
// offset of 4 or 8 bytes is applied to it by the register access functions.

array bits(32) _R[RName];

// RBankSelect()
// =============

RName RBankSelect(bits(5) mode, RName usr, RName fiq, RName irq,
                  RName svc, RName abt, RName und, RName mon)
    if BadMode(mode) then
        UNPREDICTABLE;
    else
        case mode of
            when '10000'  result = usr;  // User mode
            when '10001'  result = fiq;  // FIQ mode
            when '10010'  result = irq;  // IRQ mode
            when '10011'  result = svc;  // Supervisor mode
            when '10110'  result = mon;  // Monitor mode
            when '10111'  result = abt;  // Abort mode
            when '11011'  result = und;  // Undefined mode
            when '11111'  result = usr;  // System mode uses User mode registers
    return result;

// RfiqBankSelect()
// ================

RName RfiqBankSelect(bits(5) mode, RName usr, RName fiq)
    return RBankSelect(mode, usr, fiq, usr, usr, usr, usr, usr);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

いずれの場合も、以前のアーキテクチャバージョンでの動作は、同じ命令セット状態を維持したままの分岐です。詳細については、次のセクションを参照して下さい。

— *インターワーキング* P. 付録 G-4（ARMv6 の場合）

— *インターワーキング* P. 付録 H-5（ARMv5 と ARMv4 の場合）

ARM コアレジスタ演算の擬似コードの詳細

次の擬似コードは、汎用レジスタへのアクセスを提供します。

```
// The names of the banked core registers.

enumeration RName {RName_0usr, RName_1usr, RName_2usr, RName_3usr, RName_4usr, RName_5usr,
                   RName_6usr, RName_7usr, RName_8usr, RName_8fiq, RName_9usr, RName_9fiq,
                   RName_10usr, RName_10fiq, RName_11usr, RName_11fiq, RName_12usr, RName_12fiq,
                   RName_SPusr, RName_SPfiq, RName_SPirq, RName_SPsvc,
                   RName_SPabt, RName_SPund, RName_SPmon,
                   RName_LRusr, RName_LRfiq, RName_LRirq, RName_LRsvc,
                   RName_LRabt, RName_LRund, RName_LRmon,
                   RName_PC};

// The physical array of banked core registers.
//
// _R[RName_PC] is defined to be the address of the current instruction. The
// offset of 4 or 8 bytes is applied to it by the register access functions.

array bits(32) _R[RName];

// RBankSelect()
// =============

RName RBankSelect(bits(5) mode, RName usr, RName fiq, RName irq,
                  RName svc, RName abt, RName und, RName mon)
    if BadMode(mode) then
        UNPREDICTABLE;
    else
        case mode of
            when '10000'  result = usr;  // User mode
            when '10001'  result = fiq;  // FIQ mode
            when '10010'  result = irq;  // IRQ mode
            when '10011'  result = svc;  // Supervisor mode
            when '10110'  result = mon;  // Monitor mode
            when '10111'  result = abt;  // Abort mode
            when '11011'  result = und;  // Undefined mode
            when '11111'  result = usr;  // System mode uses User mode registers
    return result;

// RfiqBankSelect()
// ================

RName RfiqBankSelect(bits(5) mode, RName usr, RName fiq)
    return RBankSelect(mode, usr, fiq, usr, usr, usr, usr, usr);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
// LookUpRName()
// =============

RName LookUpRName(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;
    case n of
        when 0   result = RName_0usr;
        when 1   result = RName_1usr;
        when 2   result = RName_2usr;
        when 3   result = RName_3usr;
        when 4   result = RName_4usr;
        when 5   result = RName_5usr;
        when 6   result = RName_6usr;
        when 7   result = RName_7usr;
        when 8   result = RfiqBankSelect(mode, RName_8usr, RName_8fiq);
        when 9   result = RfiqBankSelect(mode, RName_9usr, RName_9fiq);
        when 10  result = RfiqBankSelect(mode, RName_10usr, RName_10fiq);
        when 11  result = RfiqBankSelect(mode, RName_11usr, RName_11fiq);
        when 12  result = RfiqBankSelect(mode, RName_12usr, RName_12fiq);
        when 13  result = RBankSelect(mode, RName_SPusr, RName_SPfiq, RName_SPirq,
                                      RName_SPsvc, RName_SPabt, RName_SPund, RName_SPmon);
        when 14  result = RBankSelect(mode, RName_LRusr, RName_LRfiq, RName_LRirq,
                                      RName_LRsvc, RName_LRabt, RName_LRund, RName_LRmon);
    return result;

// Rmode[] - non-assignment form
// =============================

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // In Non-secure state, check for attempted use of Monitor mode ('10110'), or of FIQ
    // mode ('10001') when the Security Extensions are reserving the FIQ registers. The
    // definition of UNPREDICTABLE does not permit this to be a security hole.
    if !IsSecure() && mode == '10110' then UNPREDICTABLE;
    if !IsSecure() && mode == '10001' && NSACR.RFR == '1' then UNPREDICTABLE;

    return _R[LookUpRName(n,mode)];

// Rmode[] - assignment form
// =========================

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 && n <= 14;

    // In Non-secure state, check for attempted use of Monitor mode ('10110'), or of FIQ
    // mode ('10001') when the Security Extensions are reserving the FIQ registers. The
    // definition of UNPREDICTABLE does not permit this to be a security hole.
    if !IsSecure() && mode == '10110' then UNPREDICTABLE;
    if !IsSecure() && mode == '10001' && NSACR.RFR == '1' then UNPREDICTABLE;

    // Writes of non word-aligned values to SP are only permitted in ARM state.
    if n == 13 && value<1:0> != '00' && CurrentInstrSet() != InstrSet_ARM then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
// LookUpRName()
// =============

RName LookUpRName(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;
    case n of
        when 0   result = RName_0usr;
        when 1   result = RName_1usr;
        when 2   result = RName_2usr;
        when 3   result = RName_3usr;
        when 4   result = RName_4usr;
        when 5   result = RName_5usr;
        when 6   result = RName_6usr;
        when 7   result = RName_7usr;
        when 8   result = RfiqBankSelect(mode, RName_8usr, RName_8fiq);
        when 9   result = RfiqBankSelect(mode, RName_9usr, RName_9fiq);
        when 10  result = RfiqBankSelect(mode, RName_10usr, RName_10fiq);
        when 11  result = RfiqBankSelect(mode, RName_11usr, RName_11fiq);
        when 12  result = RfiqBankSelect(mode, RName_12usr, RName_12fiq);
        when 13  result = RBankSelect(mode, RName_SPusr, RName_SPfiq, RName_SPirq,
                                      RName_SPsvc, RName_SPabt, RName_SPund, RName_SPmon);
        when 14  result = RBankSelect(mode, RName_LRusr, RName_LRfiq, RName_LRirq,
                                      RName_LRsvc, RName_LRabt, RName_LRund, RName_LRmon);
    return result;

// Rmode[] - non-assignment form
// =============================

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // In Non-secure state, check for attempted use of Monitor mode ('10110'), or of FIQ
    // mode ('10001') when the Security Extensions are reserving the FIQ registers. The
    // definition of UNPREDICTABLE does not permit this to be a security hole.
    if !IsSecure() && mode == '10110' then UNPREDICTABLE;
    if !IsSecure() && mode == '10001' && NSACR.RFR == '1' then UNPREDICTABLE;

    return _R[LookUpRName(n,mode)];

// Rmode[] - assignment form
// =========================

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 && n <= 14;

    // In Non-secure state, check for attempted use of Monitor mode ('10110'), or of FIQ
    // mode ('10001') when the Security Extensions are reserving the FIQ registers. The
    // definition of UNPREDICTABLE does not permit this to be a security hole.
    if !IsSecure() && mode == '10110' then UNPREDICTABLE;
    if !IsSecure() && mode == '10001' && NSACR.RFR == '1' then UNPREDICTABLE;

    // Writes of non word-aligned values to SP are only permitted in ARM state.
    if n == 13 && value<1:0> != '00' && CurrentInstrSet() != InstrSet_ARM then UNPREDICTABLE;
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    _R[LookUpRName(n,mode)] = value;
    return;

// R[] - non-assignment form
// =========================

bits(32) R[integer n]
    assert n >= 0 && n <= 15;
    if n == 15 then
        offset = if CurrentInstrSet() == InstrSet_ARM then 8 else 4;
        result = _R[RName_PC] + offset;
    else
        result = Rmode[n, CPSR.M];
    return result;

// R[] - assignment form
// =====================

R[integer n] = bits(32) value
    assert n >= 0 && n <= 14;
    Rmode[n, CPSR.M] = value;
    return;

// BranchTo()
// ==========

BranchTo(bits(32) address)
    _R[RName_PC] = address;
    return;
```

B1.3.3 Program Status Registers (PSRs)

The application level programmers' model provides the Application Program Status Register, see *The Application Program Status Register (APSR)* on page A2-14. This is an application level alias for the *Current Program Status Register* (CPSR). The system level view of the CPSR extends the register, adding system level information.

Each of the exception modes has its own saved copy of the CPSR, the *Saved Program Status Register* (SPSR), as shown in Figure B1-1 on page B1-9. For example, the SPSR for Monitor mode is called SPSR_mon.

The Current Program Status Register (CPSR)

The *Current Program Status Register* (CPSR) holds processor status and control information:

- the APSR, see *The Application Program Status Register (APSR)* on page A2-14
- the current instruction set state, see *ISETSTATE* on page A2-15
- the execution state bits for the Thumb If-Then instruction, see *ITSTATE* on page A2-17
- the current endianness, see *ENDIANSTATE* on page A2-19
- the current processor mode

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳


```
    _R[LookUpRName(n,mode)] = value;
    return;

// R[] - non-assignment form
// =========================

bits(32) R[integer n]
    assert n >= 0 && n <= 15;
    if n == 15 then
        offset = if CurrentInstrSet() == InstrSet_ARM then 8 else 4;
        result = _R[RName_PC] + offset;
    else
        result = Rmode[n, CPSR.M];
    return result;

// R[] - assignment form
// =====================

R[integer n] = bits(32) value
    assert n >= 0 && n <= 14;
    Rmode[n, CPSR.M] = value;
    return;

// BranchTo()
// ==========

BranchTo(bits(32) address)
    _R[RName_PC] = address;
    return;
```


B1.3.3 プログラムステータス レジスタ (PSR)

アプリケーションレベルのプログラマモデルでは、アプリケーションプログラム ステータスレジスタが提供されます。*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。このレジスタは、*カレントプログラム ステータスレジスタ* (CPSR) のアプリケーションレベルのエイリアスです。システムレベルから見た CPSR では、このレジスタが拡張され、システムレベルの情報が追加されます。

P. B1-9 の図 B1-1 に示すように、例外モードごとに、CPSR と*保存プログラムステータス レジスタ* (SPSR) の保存コピーが別に存在します。例えば、モニタモードの SPSR は SPSR_mon という名前です。

カレントプログラム ステータスレジスタ (CPSR)

カレントプログラム ステータスレジスタ (CPSR) には、次のようなプロセッサのステータスと制御情報が保持されます。

- APSR（*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照）
- 現在の命令セット状態（*ISETSTATE* P. A2-15 を参照）
- Thumb If-Then 命令の実行状態ビット（*ITSTATE* P. A2-17 を参照）
- 現在のエンディアン形式（*ENDIANSTATE* P. A2-19 を参照）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- interrupt and asynchronous abort disable bits.

The non-APSR bits of the CPSR have defined reset values. These are shown in the `TakeReset()` pseudocode function, see *Reset* on page B1-48.

Writes to the CPSR have side-effects on various aspects of processor operation. All of these side-effects, except for those on memory accesses caused by fetching instructions, are synchronous to the CPSR write. This means they are guaranteed not to be visible to earlier instructions in the execution stream, and they are guaranteed to be visible to later instructions in the execution stream.

Fetching an instruction causes an instruction fetch memory access. In addition, in a Virtual Memory System Architecture (VMSA) implementation, fetching an instruction can cause a translation table walk. The privilege of these memory accesses can be affected by changes to the mode field of the CPSR. Also, if the Security Extensions are implemented the virtual memory space of these accesses can be affected by changes to the mode field. Those mode changes take effect on the memory accesses as follows:

- A mode change by an exception entry is synchronous to the exception entry. This applies to all exception entries, including the exception entry for a synchronous exception generated directly by an instruction.
- A mode change by an exception return instruction is synchronous to the instruction.
- A mode change by an instruction other than an exception return and that is not the result of a synchronous exception generated directly by the instruction. Such a mode change can be the result of a `CPS` or `MSR` instructions, and:
  - — is guaranteed not to be visible to memory accesses caused by fetching earlier instructions in the execution stream
  - — is guaranteed to be visible to memory accesses caused by fetching instructions after the next exception entry, exception return instruction, or `ISB` instruction in the execution stream
  - — might or might not affect memory accesses caused by fetching instructions between the mode change instruction and the point where mode changes are guaranteed to be visible.

See *Exception return* on page B1-38 for the definition of exception return instructions.

The Saved Program Status Registers (SPSRs)

The purpose of an SPSR is to record the pre-exception value of the CPSR. When taking an exception, the processor copies the CPSR to the SPSR of the exception mode it is about to enter. Saving this value means the exception handler can:

- on exception return, restore the CPSR to the value it had when the exception was taken
- examine the value the CPSR had when the exception was taken, for example to determine the instruction set state in which the instruction that caused an Undefined Instruction exception was executed.

The SPSRs do not have defined reset values.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- 現在のプロセッサモード
- 割り込みディセーブルビットと非同期アボートディセーブルビット

CPSR のうち、APSR 以外のビットにはリセット時の値が定義されています。これらの値は、`TakeReset()` 擬似コード関数に示されています。*リセット* P. B1-48 を参照して下さい。

CPSR への書き込みは、プロセッサ動作のさまざまな側面に副作用をもたらします。これらの副作用は、命令フェッチによるメモリアクセスに関するものを除いて、CPSR 書き込みと同期します。つまり、これらの副作用は、実行ストリーム内でその命令よりも前にある命令からは可視でなく、その命令より後にある命令からは可視であることが保証されます。

命令フェッチによって、命令フェッチ メモリアクセスが発生します。加えて、仮想メモリシステム アーキテクチャ (VMSA) 実装では、命令フェッチによって変換テーブルウォークが発生することがあります。これらのメモリアクセスの特権は、CPSR のモードフィールドに対する変更の影響を受ける可能性があります。また、セキュリティ拡張機能が実装されている場合は、これらのアクセスの仮想メモリ空間が、モードフィールドに対する変更の影響を受ける可能性があります。これらのモード変更は、次のようなメモリアクセスに影響を与えます。

- 例外の開始によるモード変更は、例外の開始と同期します。このことは、命令によって直接生成される同期例外の開始を含む、すべての例外の開始に適用されます。
- 例外からの復帰命令によるモード変更は、命令と同期します。
- 例外からの復帰以外の命令によるモード変更、および命令によって直接生成された同期例外の結果ではないモード変更。このようなモード変更は、`CPS` 命令または `MSR` 命令によって生成される可能性があり、次のような特徴を持っています。
  - – 実行ストリーム内で、それ以前の命令フェッチによって発生したメモリアクセスから可視でないことが保証されます。
  - – 実行ストリーム内の次の例外の開始、例外からの復帰命令、または `ISB` 命令より後の命令フェッチによって発生したメモリアクセスから可視であることが保証されます。
  - – モード変更命令と、モード変更が可視になることが保証される時点との間の命令フェッチによって発生したメモリアクセスには、影響を与える場合と与えない場合があります。

例外からの復帰命令の定義については、*例外からの復帰* P. B1-38 を参照して下さい。

保存プログラム ステータスレジスタ (SPSR)

SPSR の目的は、CPSR の例外前の値を記録することです。例外の取得時に、プロセッサは、遷移先の例外モードの SPSR に CPSR をコピーします。この値の保存により、例外ハンドラは次の操作を実行できます。

- 例外からの復帰時に、例外が取得された時点の値に CPSR を復元する。
- 例外が取得された時点の CPSR の値を検査し、例えば、未定義命令例外を引き起こした命令が実行された命令セット状態を特定する。

SPSR のリセット値は定義されていません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Format of the CPSR and SPSRs

The format of the CPSR and SPSRs is:

31	30	29	28	27	26 25	24	23 20	19 16	15 10	9	8	7	6	5	4 0
N	Z	C	V	Q	IT [1:0]	J	Reserved	GE[3:0]	IT[7:2]	E	A	I	F	T	M[4:0]

Condition code flags, bits [31:28]

Set on the result of instruction execution. The flags are:

N, bit [31] Negative condition code flag

Z, bit [30] Zero condition code flag

C, bit [29] Carry condition code flag

V, bit [28] Overflow condition code flag.

The condition code flags can be read or written in any mode, and are described in *The Application Program Status Register (APSR)* on page A2-14.

Q, bit [27] Cumulative saturation flag. This flag can be read or written in any mode, and is described in *The Application Program Status Register (APSR)* on page A2-14.

IT[7:0], bits [15:10,26:25]

If-Then execution state bits for the Thumb `IT` (If-Then) instruction. *ITSTATE* on page A2-17 describes the encoding of these bits. CPSR.IT[7:0] are the IT[7:0] bits described there. For more information, see *IT* on page A8-104.

For details of how these bits can be accessed see *Accessing the execution state bits* on page B1-18.

J, bit [24] Jazelle bit, see the description of the T bit, bit [5].

Bits [23:20] Reserved. RAZ/SBZP.

GE[3:0], bits [19:16]

Greater than or Equal flags, for SIMD instructions.

The GE[3:0] field can be read or written in any mode, and is described in *The Application Program Status Register (APSR)* on page A2-14.

E, bit [9] Endianness execution state bit. Controls the load and store endianness for data accesses:

0 Little endian operation

1 Big endian operation.

This bit is ignored by instruction fetches.

ENDIANSTATE on page A2-19 describes the encoding of this bit. CPSR.E is the ENDIANSTATE bit described there.

For details of how this bit can be accessed see *Accessing the execution state bits* on page B1-18.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

CPSR と SPSR のフォーマット

CPSR と SPSR のフォーマットは次のとおりです。

31	30	29	28	27	26 25	24	23 20	19 16	15 10	9	8	7	6	5	4 0
N	Z	C	V	Q	IT [1:0]	J	予約	GE[3:0]	IT[7:2]	E	A	I	F	T	M[4:0]

条件コードフラグ、ビット [31:28]

命令実行の結果にセットされます。フラグは次のとおりです。

N、ビット [31] 負の条件コードフラグ

Z、ビット [30] 0 の条件コードフラグ

C、ビット [29] キャリーの条件コードフラグ

V、ビット [28] オーバフローの条件コードフラグ

条件コードフラグは任意のモードで読み書きすることができます。*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。

Q、ビット [27] 累積飽和フラグ。このフラグは、任意のモードで読み書きすることができます。*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。

IT[7:0]、ビット [15:10、26:25]

Thumb `IT`(If-Then) 命令の If-Then 実行状態ビット。これらのビットのエンコードについては、*ITSTATE* P. A2-17 を参照して下さい。CPSR.IT[7:0] は、ここで説明されている IT[7:0] です。詳細については、*IT* P. A8-104 を参照して下さい。

これらのビットへのアクセス方法については、*実行状態ビットへのアクセス* P. B1-18 を参照して下さい。

J、ビット [24] Jazelle ビット。T ビット（ビット [5]）の説明を参照して下さい。

ビット [23:20] 予約。RAZ/SBZP。

GE[3:0]、ビット [19:16]

以上フラグ、SIMD 命令用。

GE[3:0] フィールドは、任意のモードで読み書きすることができます。*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。

E、ビット [9] エンディアン形式実行状態ビット。データアクセスのロードおよびストアのエンディアン形式を制御します。

0 リトルエンディアン動作

1 ビッグエンディアン動作

このビットは、命令フェッチでは無視されます。

このビットのエンコードを、*ENDIANSTATE* P. A2-19 に示します。CPSR.E は、ここで説明されている ENDIANSTATE ビットです。

このビットへのアクセス方法については、*実行状態ビットへのアクセス* P. B1-18 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Mask bits, bits [8:6]

The mask bits disable some asynchronous exceptions. The three mask bits are:

A, bit [8] Asynchronous abort disable bit. Used to mask asynchronous aborts.

I, bit [7] Interrupt disable bit. Used to mask IRQ interrupts.

F, bit [6] Fast interrupt disable bit. Used to mask FIQ interrupts.

The possible values of each bit are:

0 Exception enabled

1 Exception disabled.

The mask bits can be written only in privileged modes. Their values can be read in any mode, but use of their values and attempts to change them by User mode code are deprecated.

Updates to the F bit are restricted if Non-maskable Fast Interrupts (NMFIs) are supported, see *Non-maskable fast interrupts* on page B1-18.

If implemented, the Security Extensions can restrict updates to the A and F bits from the Non-secure state, see *Use of the A, F, and Mode bits by the Security Extensions* on page B1-19.

T, bit [5] Thumb execution state bit. This bit and the J execution state bit, bit [24], determine the instruction set state of the processor, ARM, Thumb, Jazelle, or ThumbEE. *ISETSTATE* on page A2-15 describes the encoding of these bits. CPSR.J and CPSR.T are the same bits as ISETSTATE.J and ISETSTATE.T respectively. For more information, see *Instruction set states* on page B1-23.

For details of how these bits can be accessed see *Accessing the execution state bits* on page B1-18.

M[4:0], bits [4:0]

Mode field. This field determines the current mode of the processor. The permitted values of this field are listed in Table B1-1 on page B1-6. All other values of M[4:0] are reserved. The effect of setting M[4:0] to a reserved value is UNPREDICTABLE.

For more information about the processor modes see *ARM processor modes* on page B1-6. Figure B1-1 on page B1-9 shows the registers that can be accessed in each mode.

This field can be written only in privileged modes. Its value can be read in any mode, but use of its value and attempts to change it by User mode code are deprecated.

If implemented, the Security Extensions restrict use of the mode field to enter Monitor and FIQ modes, see *Use of the A, F, and Mode bits by the Security Extensions* on page B1-19.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

マスクビット、ビット [8:6]

マスクビットは、一部の非同期例外を無効にします。次の 3 つのマスクビットがあります。

A、ビット [8] 非同期アボート ディセーブルビット。非同期アボートをマスクするために使用されます。

I、ビット [7] 割り込みディセーブルビット。IRQ 割り込みをマスクするために使用されます。

F、ビット [6] 高速割り込みディセーブルビット。FIQ 割り込みをマスクするために使用されます。

各ビットの可能な値は次のとおりです。

0 例外が可能な場合

1 例外が不可能な場合

マスクビットは特権モードでのみ書き込むことができます。この値は任意のモードで読み出すことができますが、ユーザモード コードによるこれらの値の使用および変更の試みは推奨されません。

マスク不能高速割り込み (NMFI) がサポートされている場合は、F ビットの更新が制限されます。*マスク不能高速割り込み* P. B1-18 を参照して下さい。

セキュリティ拡張機能が実装されている場合は、非セキュア状態からの A ビットと F ビットの更新を制限することができます。*セキュリティ拡張機能による A、F、Mode の各ビットの使用* P. B1-19 を参照して下さい。

T、ビット [5] Thumb 実行状態ビット。このビットと J 実行状態ビット（ビット [24]）によって、プロセッサの命令セット状態が ARM、Thumb、Jazelle、ThumbEE のいずれかに決定されます。これらのビットのエンコードについては、*ISETSTATE* P. A2-15 を参照して下さい。CPSR.J と CPSR.T は、それぞれ ISETSTATE.J および ISETSTATE.T と同じビットです。詳細については、*命令セット状態* P. B1-23 を参照して下さい。

これらのビットへのアクセス方法については、*実行状態ビットへのアクセス* P. B1-18 を参照して下さい。

M[4:0]、ビット [4:0]

モードフィールド。このフィールドによって、現在のプロセッサモードが決定されます。このフィールドに設定可能な値を、P. B1-6 の表 B1-1 に示します。その他の M[4:0] の値はすべて予約済みです。M[4:0] を予約済みの値に設定した場合の動作は予測不能です。

プロセッサモードの詳細については、*ARM プロセッサモード* P. B1-6 を参照して下さい。各モードでアクセス可能なレジスタを P. B1-9 の図 B1-1 に示します。

このフィールドは、特権モードでのみ書き込むことができます。この値は任意のモードで読み出すことができますが、ユーザモードのコードでこの値を使用すること、および変更の試みは推奨されません。

セキュリティ拡張機能が実装されている場合は、モニタモードと FIQ モードに遷移するためのモードフィールドの使用が制限されます。*セキュリティ拡張機能による A、F、Mode の各ビットの使用* P. B1-19 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the execution state bits

The execution state bits are the IT[7:0], J, E, and T bits. In exception modes you can read or write these bits in the current SPSR.

In the CPSR, unless the processor is in Debug state:

- The execution state bits, other than the E bit, are RAZ when read by an `MRS` instruction.
- Writes to the execution state bits, other than the E bit, by an `MSR` instruction are:
  - — For ARMv7 and ARMv6T2, ignored in all modes.
  - — For architecture variants before ARMv6T2, ignored in User mode and required to write zeros in privileged modes. If a nonzero value is written in a privileged mode, behavior is UNPREDICTABLE.

Instructions other than `MRS` and `MSR` that access the execution state bits can read and write them in any mode.

Unlike the other execution state bits in the CPSR, CPSR.E can be read by an `MRS` instruction and written by an `MSR` instruction. However, using the CPSR.E value read by an `MRS` instruction is deprecated, and using an `MSR` instruction to change the value of CPSR.E is deprecated.

Note

- Use the `SETEND` instruction to change the current endianness.
- To determine the current endianness, use an `LDR` instruction to load a word of memory whose value is known and will differ if the endianness is reversed. For example, use an `LDR` (literal) instruction to load a word whose four bytes are `0x01`, `0x00`, `0x00`, and `0x00` in ascending order of memory address. The `LDR` instruction loads the destination register with:
  - — `0x00000001` if the current endianness is little-endian
  - — `0x01000000` if the current endianness is big-endian.

For more information about the behavior of these bits in Debug state see *Behavior of the PC and CPSR in Debug state* on page C5-7.

Non-maskable fast interrupts

Exceptions, debug events and checks on page A2-81 introduces the two levels of external interrupts to an ARM processor, Interrupt Requests or IRQs and higher priority Fast Interrupt Requests or FIQs. Both IRQs and FIQs can be masked by bits in the CPSR, see *Program Status Registers (PSRs)* on page B1-14:

- when the CPSR.I bit is set to 1, IRQ interrupts are masked
- when the CPSR.F bit is set to 1, FIQ interrupts are masked.

ARMv7 supports an operating mode where FIQs are not maskable by software. This *Non-maskable Fast Interrupt* (NMFI) operation is controlled by a configuration input signal to the processor, that is asserted HIGH to enable NMFI operation. There is no software control of NMFI operation.

Software can detect whether FIQs are maskable by reading the SCTLR.NMFI bit:

NMFI == 0 Software can mask FIQs by setting the CPSR.F bit to 1

NMFI == 1 Software cannot mask FIQs.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

実行状態ビットへのアクセス

実行状態ビットは、IT[7:0]、J、E、T の各ビットです。例外モードで、現在の SPSR 内のこれらのビットを読み書きすることができます。

CPSR では、プロセッサがデバッグ状態でない場合、次の規則が適用されます。

- E ビット以外の実行状態ビットは、`MRS` 命令によって RAZ として読み出されます。
- E ビット以外の実行状態ビットへ `MSR` 命令によって書き込む場合、次の規則が適用されます。
  - – ARMv7 と ARMv6T2 の場合は、すべてのモードで無視されます。
  - – ARMv6T2 以前のアーキテクチャバリアントの場合は、ユーザモードでは無視され、特権モードでは 0 を書き込む必要があります。特権モードで 0 以外の値が書き込まれた場合の動作は、予測不能です。

実行状態ビットにアクセスする `MRS` と `MSR` 以外の命令では、任意のモードで実行状態ビットを読み書きすることができます。

CPSR 内のその他の実行状態ビットと違って、CPSR.E は、`MRS` 命令で読み出して、`MSR` 命令で書き込むことができます。ただし、`MRS` 命令で読み出した CPSR.E 値の使用と、CPSR.E の値を変更するための `MSR` 命令の使用は推奨されません。

注

- 現在のエンディアン形式を変更するには `SETEND` 命令を使用して下さい。
- 現在のエンディアン形式を特定するには、値が既知であるメモリのワードを `LDR` 命令でロードします。エンディアン形式が逆であれば、ロードされる値は異なります。例えば、`LDR`（リテラル）命令を使用して、メモリアドレスの昇順に `0x01`、`0x00`、`0x00`、`0x00` の 4 バイトのワードをロードします。`LDR` 命令によって、デスティネーションレジスタに次の値がロードされます。
  - – `0x00000001`（現在のエンディアン形式がリトルエンディアンの場合）
  - – `0x01000000`（現在のエンディアン形式がビッグエンディアンの場合）

デバッグ状態でのこれらのビットの動作については、*デバッグ状態での PC および CPSR の動作* P. C5-7 を参照して下さい。

マスク不能高速割り込み

例外、デバッグイベント、チェック P. A2-81 では、ARM プロセッサへの 2 レベルの外部割り込みとして、IRQ（割り込み要求）と、より優先度の高い FIQ（高速割り込み要求）が紹介されています。IRQ と FIQ はどちらも、CPSR 内のビットによりマスクすることができます。*プログラムステータス レジスタ (PSR)* P. B1-14 を参照して下さい。

- CPSR.I ビットが 1 にセットされた場合は、IRQ 割り込みがマスクされます。
- CPSR.F ビットが 1 にセットされた場合は、FIQ 割り込みがマスクされます。

ARMv7 では、ソフトウェアで FIQ をマスクできない動作モードがサポートされています。この *マスク不能高速割り込み* (NMFI) 動作は、NMFI 動作を有効にするために HIGH にアサートされる、プロセッサへの構成入力信号によって制御されます。NMFI 動作はソフトウェアで制御できません。

ソフトウェアで SCTLR.NMFI ビットを読み出すことによって、FIQ がマスク可能かどうかを検出することができます。

NMFI = 0　ソフトウェアで CPSR.F ビットを 1 にセットして、FIQ をマスクできます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For more information see:

- *c1, System Control Register (SCTLR)* on page B3-96 for a VMSA implementation
- *c1, System Control Register (SCTLR)* on page B4-45 for a PMSA implementation.

It is IMPLEMENTATION DEFINED whether an ARMv7 processor supports NMFIs. The SCTLR.NMFI bit is RAO only if the processor supports NMFIs and the configuration input signal is asserted HIGH, otherwise it is RAZ.

When the SCTLR.NMFI bit is 1:

- an instruction writing 0 to the CPSR.F bit clears it to 0, but an instruction attempting to write 1 to it leaves it unchanged.
- CPSR.F can be set to 1 only by exception entries, as described in *CPSR M field and A, I, and F mask bit values on exception entry* on page B1-36.

Use of the A, F, and Mode bits by the Security Extensions

When the Security Extensions are implemented and the processor is in the Non-secure state:

- the CPSR.F bit cannot be changed if the SCR.FW bit is set to 0
- the CPSR.A bit cannot be changed if the SCR.AW bit is set to 0
- the effect of setting CPSR.M to 0b10110, Monitor mode, is UNPREDICTABLE
- the effect of setting CPSR.M to 0b10001, FIQ mode, is UNPREDICTABLE if NSACR.RFR is set to 1.

Note

- When the Security Extensions are implemented and the processor is in the Non-secure state the SPSR.F and SPSR.A bits can be changed even if the corresponding bits in the SCR are set to 0. However, when the SPSR is copied to the CPSR the CPSR.F and CPSR.A bits are not updated if the corresponding bits in the SCR are set to 0.
- UNPREDICTABLE behavior must not be a security hole. Therefore, every implementation must ensure that:
  - If NSACR.RFR is 0, setting CPSR.M to 0b10110 when in Non-secure state cannot cause entry to either Monitor mode or Secure state
  - If NSACR.RFR is 1, setting CPSR.M to 0b10001 or 0b10110 when in Non-secure state cannot cause entry to Monitor mode, FIQ mode or Secure state.

For more information about the access controls provided by the Security Extensions see *c1, Secure Configuration Register (SCR)* on page B3-106.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

NMFI == 1　ソフトウェアで FIQ をマスクすることはできません。

詳細については、次のセクションを参照して下さい。

- *c1、システム制御レジスタ (SCTLR)* P. B3-96（VMSA 実装の場合）
- *c1、システム制御レジスタ (SCTLR)* P. B4-45 （PMSA 実装の場合）

ARMv7 プロセッサで NMFI がサポートされるかどうかは、実装定義です。プロセッサで NMFI がサポートされており、構成入力信号が HIGH にアサートされている場合にのみ、SCTLR.NMFI ビットが RAO になり、それ以外は RAZ になります。

SCTLR.NMFI ビットが 1 の場合、次の規則が適用されます。

- CPSR.F ビットに 0 を書き込む命令では 0 にクリアされますが、このビットに 1 を書き込む命令では変更されません。
- *例外の開始時点の CPSR M フィールドと、A、I、F の各マスクビットの値* P. B1-36 で説明されているように、CPSR.F は例外の開始でのみ 1 に設定することができます。

セキュリティ拡張機能による A、F、Mode の各ビットの使用

セキュリティ拡張機能が実装されており、プロセッサが非セキュア状態にある場合、次の規則が適用されます。

- SCR.FW ビットが 0 にセットされている場合は、CPSR.F ビットを変更することができません。
- SCR.AW ビットが 0 にセットされている場合は、CPSR.A ビットを変更することができません。
- CPSR.M を 0b10110 のモニタモードにセットした場合の動作は、予測不能です。
- NSACR.RFR が 1 にセットされており、CPSR.M を 0b10001 の FIQ モードにセットした場合の動作は、予測不能です。

注

- セキュリティ拡張機能が実装されており、プロセッサが非セキュア状態にある場合は、SCR 内の対応するビットが 0 にセットされていた場合でも、SPSR.F ビットと SPSR.A ビットを変更することができます。ただし、SPSR を CPSR にコピーしても、SCR 内の対応するビットが 0 にセットされている場合は、CPSR.F ビットと CPSR.A ビットは更新されません。
- 予測不能動作をセキュリティホールにしないようにする必要があります。そのためには、すべての実装で次の条件を保証する必要があります。
  - NSACR.RFR が 0 の場合は、非セキュア状態で CPSR.M を 0b10110 にセットしても、モニタモードまたはセキュア状態に遷移させることができない。
  - NSACR.RFR が 1 の場合は、非セキュア状態で CPSR.M を 0b10001 または 0b10110 にセットしても、モニタモード、FIQ モード、またはセキュア状態に遷移させることができない。

セキュリティ拡張機能で提供されるアクセス制御の詳細については、*c1、セキュア構成レジスタ (SCR)* P. B3-106 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Software running in Non-secure state might not be able to set the CPSR.F bit to 1 to mask FIQs, as described in *Use of the A, F, and Mode bits by the Security Extensions* on page B1-19. Table B1-2 shows how the SCTLR.NMFI bit interacts with the SCR.FW bit to control access to the CPSR.F bit, in the Secure and Non-secure security states.

Table B1-2 Summary of NMFI behavior when Security Extensions are implemented

Security state	SCR.FW bit	SCTLR.NMFI bit	CPSR.F bit properties
Secure	x	0	F bit can be written to 0 or 1
		1	F bit can be written to 0 but not to 1
Non-secure	0	x	F bit cannot be written
	1	0	F bit can be written to 0 or 1
		1	F bit can be written to 0 but not to 1

Note

The SCTLR.NMFI bit is common to the Secure and Non-secure versions of the SCTLR, because it is a read-only bit that reflects the value of a configuration input signal.

Pseudocode details of PSR operations

The following pseudocode gives access to the PSRs:

```
bits(32) CPSR, SPSR_fiq, SPSR_irq, SPSR_svc, SPSR_mon, SPSR_abt, SPSR_und;

// SPSR[] - non-assignment form
// ============================

bits(32) SPSR[]
    if BadMode(CPSR.M) then
        UNPREDICTABLE;
    else
        case CPSR.M of
            when '10001'  result = SPSR_fiq;  // FIQ mode
            when '10010'  result = SPSR_irq;  // IRQ mode
            when '10011'  result = SPSR_svc;  // Supervisor mode
            when '10110'  result = SPSR_mon;  // Monitor mode
            when '10111'  result = SPSR_abt;  // Abort mode
            when '11011'  result = SPSR_und;  // Undefined mode
            otherwise     UNPREDICTABLE;
    return result;

// SPSR[] - assignment form
// ========================
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

セキュリティ拡張機能による A、F、Mode の各ビットの使用 P. B1-19 で説明されているように、非セキュア状態で実行中のソフトウェアは、CPSR.F ビットを 1 にセットして FIQ をマスクできない可能性があります。セキュアと非セキュアのセキュリティ状態において、SCTLR.NMFI ビットと SCR.FW ビットがどのように相互作用して CPSR.F ビットへのアクセスが制御されるかを、表 B1-2 に示します。

表 B1-2 セキュリティ拡張機能が実装されている場合の NMFI 動作の概要

セキュリティ状態	SCR.FW ビット	SCTLR.NMFI ビット	CPSR.F ビットのプロパティ
セキュア	x	0	F ビットには 0 または 1 を書き込むことができます。
		1	F ビットには 0 を書き込むことはできますが、1 を書き込むことはできません。
非セキュア	0	x	F ビットには書き込めません。
	1	0	F ビットには 0 または 1 を書き込むことができます。
		1	F ビットには 0 を書き込むことはできますが、1 を書き込むことはできません。

注

SCTLR.NMFI ビットは、構成入力信号の値を反映した読み出し専用ビットであるため、セキュアバージョンと非セキュアバージョンの SCTLR で共通しています。

PSR 動作の擬似コードの詳細

次の擬似コードは、PSR へのアクセスを提供します。

```
bits(32) CPSR, SPSR_fiq, SPSR_irq, SPSR_svc, SPSR_mon, SPSR_abt, SPSR_und;

// SPSR[] - non-assignment form
// ============================

bits(32) SPSR[]
    if BadMode(CPSR.M) then
        UNPREDICTABLE;
    else
        case CPSR.M of
            when '10001'  result = SPSR_fiq;  // FIQ mode
            when '10010'  result = SPSR_irq;  // IRQ mode
            when '10011'  result = SPSR_svc;  // Supervisor mode
            when '10110'  result = SPSR_mon;  // Monitor mode
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
SPSR[] = bits(32) value
    if BadMode(CPSR.M) then
        UNPREDICTABLE;
    else
        case CPSR.M of
            when '10001'  SPSR_fiq = value;  // FIQ mode
            when '10010'  SPSR_irq = value;  // IRQ mode
            when '10011'  SPSR_svc = value;  // Supervisor mode
            when '10110'  SPSR_mon = value;  // Monitor mode
            when '10111'  SPSR_abt = value;  // Abort mode
            when '11011'  SPSR_und = value;  // Undefined mode
            otherwise     UNPREDICTABLE;
    return;

// CPSRWriteByInstr()
// ==================

CPSRWriteByInstr(bits(32) value, bits(4) bytemask, boolean affect_execstate)

    privileged = CurrentModeIsPrivileged();
    nmfi = (SCTLR.NMFI == '1');

    if bytemask<3> == '1' then
        CPSR<31:27> = value<31:27>;             // N,Z,C,V,Q flags
        if affect_execstate then
            CPSR<26:24> = value<26:24>;         // IT<1:0>,J execution state bits

    if bytemask<2> == '1' then
        // bits <23:20> are reserved SBZP bits
        CPSR<19:16> = value<19:16>;             // GE<3:0> flags

    if bytemask<1> == '1' then
        if affect_execstate then
            CPSR<15:10> = value<15:10>;         // IT<7:2> execution state bits
        CPSR<9> = value<9>;                     // E bit is user-writable
        if privileged && (IsSecure() || SCR.AW == '1') then
            CPSR<8> = value<8>;                 // A interrupt mask

    if bytemask<0> == '1' then
        if privileged then
            CPSR<7> = value<7>;                 // I interrupt mask
        if privileged && (IsSecure() || SCR.FW == '1') && (!nmfi || value<6> == '0') then
            CPSR<6> = value<6>;                 // F interrupt mask
        if affect_execstate then
            CPSR<5> = value<5>;                 // T execution state bit
        if privileged then
            if BadMode(value<4:0>) then
                UNPREDICTABLE;
            else
                // Check for attempts to enter modes only permitted in Secure state from
                // Non-secure state. These are Monitor mode ('10110'), and FIQ mode ('10001')
                // if the Security Extensions have reserved it. The definition of UNPREDICTABLE
                // does not permit the resulting behavior to be a security hole.
                if !IsSecure() && value<4:0> == '10110' then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
            when '10111'  result = SPSR_abt;  // Abort mode
            when '11011'  result = SPSR_und;  // Undefined mode
            otherwise     UNPREDICTABLE;
    return result;

// SPSR[] - assignment form
// ========================

SPSR[] = bits(32) value
    if BadMode(CPSR.M) then
        UNPREDICTABLE;
    else
        case CPSR.M of
            when '10001'  SPSR_fiq = value;  // FIQ mode
            when '10010'  SPSR_irq = value;  // IRQ mode
            when '10011'  SPSR_svc = value;  // Supervisor mode
            when '10110'  SPSR_mon = value;  // Monitor mode
            when '10111'  SPSR_abt = value;  // Abort mode
            when '11011'  SPSR_und = value;  // Undefined mode
            otherwise     UNPREDICTABLE;
    return;

// CPSRWriteByInstr()
// ==================

CPSRWriteByInstr(bits(32) value, bits(4) bytemask, boolean affect_execstate)

    privileged = CurrentModeIsPrivileged();
    nmfi = (SCTLR.NMFI == '1');

    if bytemask<3> == '1' then
        CPSR<31:27> = value<31:27>;             // N,Z,C,V,Q flags
        if affect_execstate then
            CPSR<26:24> = value<26:24>;         // IT<1:0>,J execution state bits

    if bytemask<2> == '1' then
        // bits <23:20> are reserved SBZP bits
        CPSR<19:16> = value<19:16>;             // GE<3:0> flags

    if bytemask<1> == '1' then
        if affect_execstate then
            CPSR<15:10> = value<15:10>;         // IT<7:2> execution state bits
        CPSR<9> = value<9>;                     // E bit is user-writable
        if privileged && (IsSecure() || SCR.AW == '1') then
            CPSR<8> = value<8>;                 // A interrupt mask

    if bytemask<0> == '1' then
        if privileged then
            CPSR<7> = value<7>;                 // I interrupt mask
        if privileged && (IsSecure() || SCR.FW == '1') && (!nmfi || value<6> == '0') then
            CPSR<6> = value<6>;                 // F interrupt mask
        if affect_execstate then
            CPSR<5> = value<5>;                 // T execution state bit
        if privileged then
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
                if !IsSecure() && value<4:0> == '10001' && NSACR.RFR == '1' then UNPREDICTABLE;
                CPSR<4:0> = value<4:0>;         // M<4:0> mode bits
    return;

// SPSRWriteByInstr()
// ==================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

    if CurrentModeIsUserOrSystem() then UNPREDICTABLE;

    if bytemask<3> == '1' then
        SPSR[]<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT<1:0>,J execution state bits

    if bytemask<2> == '1' then
        // bits <23:20> are reserved SBZP bits
        SPSR[]<19:16> = value<19:16>;  // GE<3:0> flags

    if bytemask<1> == '1' then
        SPSR[]<15:8> = value<15:8>;    // IT<7:2> execution state bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        SPSR[]<7:5> = value<7:5>;      // I,F interrupt masks, T execution state bit
        if BadMode(value<4:0>) then    // Mode bits
            UNPREDICTABLE;
        else
            SPSR[]<4:0> = value<4:0>;

    return;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
                if BadMode(value<4:0>) then
                    UNPREDICTABLE;
                else
                    // Check for attempts to enter modes only permitted in Secure state from
                    // Non-secure state. These are Monitor mode ('10110'), and FIQ mode ('10001')
                    // if the Security Extensions have reserved it. The definition of UNPREDICTABLE
                    // does not permit the resulting behavior to be a security hole.
                    if !IsSecure() && value<4:0> == '10110' then UNPREDICTABLE;
                    if !IsSecure() && value<4:0> == '10001' && NSACR.RFR == '1' then UNPREDICTABLE;
                    CPSR<4:0> = value<4:0>;         // M<4:0> mode bits
    return;

// SPSRWriteByInstr()
// ==================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

    if CurrentModeIsUserOrSystem() then UNPREDICTABLE;

    if bytemask<3> == '1' then
        SPSR[]<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT<1:0>,J execution state bits

    if bytemask<2> == '1' then
        // bits <23:20> are reserved SBZP bits
        SPSR[]<19:16> = value<19:16>;  // GE<3:0> flags

    if bytemask<1> == '1' then
        SPSR[]<15:8> = value<15:8>;    // IT<7:2> execution state bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        SPSR[]<7:5> = value<7:5>;      // I,F interrupt masks, T execution state bit
        if BadMode(value<4:0>) then    // Mode bits
            UNPREDICTABLE;
        else
            SPSR[]<4:0> = value<4:0>;

    return;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.4 Instruction set states

The instruction set states are described in Chapter A2 *Application Level Programmers' Model* and application level operations on them are described there. This section supplies more information about how they interact with system level functionality, in the sections:

- *Exceptions and instruction set state*.
- *Unimplemented instruction sets*.

B1.4.1 Exceptions and instruction set state

An exception is handled in the appropriate exception mode. The SCTLR.TE bit determines the processor instruction set state that handles exceptions. If necessary, the processor changes to this instruction set state on exception entry. For more information see:

- *c1, System Control Register (SCTLR)* on page B3-96 for a VMSA implementation
- *c1, System Control Register (SCTLR)* on page B4-45 for a PMSA implementation.

When an exception is taken, the value of the CPSR before the exception is written to the SPSR for the exception mode.

On returning from the exception:

- the CPSR is restored:
  - — from a memory location if the RFE instruction is used
  - — otherwise, from the SPSR for the exception mode
- the processor instruction set state is determined by the restored CPSR.J and CPSR.T values.

Note

The Reset exception is a special case and behaves differently, see *Reset* on page B1-48.

B1.4.2 Unimplemented instruction sets

The CPSR.J and CPSR.T bits define the current instruction set state, see *ISETSTATE* on page A2-15. The Jazelle state is optional, and the ThumbEE state is optional in the ARMv7-R architecture. Some system instructions permit an attempt to set CPSR.J and CPSR.T to values that select an unimplemented instruction set option, for example to set CPSR.J = 1, CPSR.T = 0 on an processor that does not implement the Jazelle state. If such values are written to CPSR.J and CPSR.T, the implementation behaves in one of these ways:

- Sets CPSR.J and CPSR.T to the requested values and causes the next instruction to be UNDEFINED.

  Entry to the Undefined Instruction handler forces the processor into the state indicated by the SCTLR.TE bit. The handler can detect the cause of the exception because CPSR.J and CPSR.T are set to the unimplemented combination in SPSR_und. Table B1-4 on page B1-34 shows the value in LR_und on exception entry.

  For the description of the SCTLR see:

  — *c1, System Control Register (SCTLR)* on page B3-96 for a VMSA implementation

  — *c1, System Control Register (SCTLR)* on page B4-45 for a PMSA implementation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.4 命令セット状態

命令セット状態の説明については、第 A2 章*「アプリケーションレベルのプログラマモデル」*を参照して下さい。ここでは、命令セット状態でのアプリケーションレベルの動作について説明しています。次のセクションで、命令セット状態がシステムレベルの機能にどのように影響するかについて説明します。

- *例外と命令セット状態*
- *未実装の命令セット*

B1.4.1 例外と命令セット状態

例外は、該当する例外モードで処理されます。SCTLR.TE ビットによって、例外を処理するプロセッサの命令セット状態が決定されます。必要に応じて、例外の開始時にプロセッサがでこの命令セット状態に遷移します。詳細については、次のセクションを参照して下さい。

- *c1、システム制御レジスタ (SCTLR)* P. B3-96（VMSA 実装の場合）
- *c1、システム制御レジスタ (SCTLR)* P. B4-45 （PMSA 実装の場合）

例外が取得されると、例外直前の CPSR の値が、例外モード用の SPSR に書き込まれます。

例外から復帰するときには、次の処理が行われます。

- CPSR が次のように復元されます。
  - – RFE 命令が使用された場合はメモリ位置から
  - – それ以外の場合は、例外モード用の SPSR から
- プロセッサの命令セット状態は、復元された CPSR.J と CPSR.T の値によって決定されます。

注

リセット例外は特殊なケースであり、動作が異なります。*リセット* P. B1-48 を参照して下さい。

B1.4.2 未実装の命令セット

CPSR.J ビットと CPSR.T ビットによって、現在の命令セット状態が定義されます。*ISETSTATE* P. A2-15 を参照して下さい。Jazelle 状態はオプションです。ARMv7-R アーキテクチャでは、ThumbEE 状態がオプションです。一部のシステム命令では、CPSR.J および CPSR.T を、実装されていない命令セットオプションを選択する値に設定する試みが許されます。例えば、Jazelle 状態が実装されていないプロセッサ上で CPSR.J = 1 と CPSR.T = 0 をセットする試みが許可されます。このような値が CPSR.J と CPSR.T に書き込まれた場合、実装は次のいずれかの動作を行います。

- CPSR.J と CPSR.T が要求された値に設定され、その次の命令が未定義になります。

  未定義命令ハンドラの開始によって、SCTLR.TE ビットで示される状態にプロセッサが強制的に遷移します。CPSR.J と CPSR.T が SPSR_und 内の未実装の組み合わせに設定されるため、ハンドラで例外の原因を検出することができます。例外の開始時点の LR_und の値を、P. B1-34 の表 B1-4 に示します。

  SCTLR の説明については、次のセクションを参照して下さい。

  – *c1、システム制御レジスタ (SCTLR)* P. B3-96（VMSA 実装の場合）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- Does not set CPSR.J and CPSR.T to the requested values. The processor might change the value of one or both of the bits in such a way that the new values correspond to an implemented instruction set state. If this is done then the instruction set state changes to this new state. The detailed behavior of the attempt to change to an unimplemented state is IMPLEMENTATION DEFINED.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

– *c1、システム制御レジスタ (SCTLR)* P. B4-45 （PMSA 実装の場合）

- CPSR.J と CPSR.T を、要求された値にセットしません。プロセッサによって、新しい値が実装されている命令セット状態に対応するように、ビットのどちらかまたは両方の値が変更される可能性があります。これが行われた場合は、命令セット状態がこの新しい状態に変化します。未実装の状態へ変更を試みた場合の動作の詳細は実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.5 The Security Extensions

It is IMPLEMENTATION DEFINED whether an ARMv7-A system includes the Security Extensions. When implemented, the Security Extensions integrate hardware security features into the architecture, to facilitate the development of secure applications. Many features of the architecture are extended to integrate with the Security Extensions, and because of this integration of the Security Extensions into the architecture, features of the Security Extensions are described in many sections of this manual.

Note

The Security Extensions are also permitted as an extension to the ARMv6K architecture. The resulting combination is sometimes known as the ARMv6Z or ARMv6KZ architecture.

General information about the Security Extensions is given in:

- *Security states*
- *Impact of the Security Extensions on the modes and exception model* on page B1-28
- *Effect of the Security Extensions on the CP15 registers* on page B3-71.

B1.5.1 Security states

The Security Extensions define two security states, Secure state and Non-secure state. All code execution takes place either in Secure state or in Non-secure state:

- each security state operates in its own virtual memory address space
- many system controls can be set independently in each of the security states
- all of the processor modes that are available in a system that does not implement the Security Extensions are available in each of the security states.

The Security Extensions also define an additional processor mode, Monitor mode, that provides a bridge between code running in Non-secure state and code running in Secure state.

The following features mean the two security states can provide more security than is typically provided by systems using the split between privileged and unprivileged code:

- the memory system provides mechanisms that prevent the Non-secure state accessing regions of the physical memory designated as Secure
- system controls that apply to the Secure state are not accessible from the Non-secure state
- entry to the Secure state from the Non-secure state is provided only by a small number of exceptions
- exit from the Secure state to the Non-secure state is provided only by a small number of mechanisms
- many operating system exceptions can be handled without changing security state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.5 セキュリティ拡張機能

ARMv7-A システムにセキュリティ拡張機能が含まれているかどうかは実装定義です。セキュリティ拡張機能が実装されている場合は、ハードウェアセキュリティ機能がアーキテクチャに統合され、セキュアなアプリケーションの開発が容易になります。セキュリティ拡張機能を統合するために、多くのアーキテクチャの機能が拡張されています。このセキュリティ拡張機能のアーキテクチャへの統合の結果、本書の多くのセクションにセキュリティ拡張機能に関する記述が含まれています。

注

セキュリティ拡張機能は、ARMv6K アーキテクチャに対する拡張として使用することもできます。この組み合わせは、ARMv6Z または ARMv6KZ アーキテクチャと呼ばれることがあります。

セキュリティ拡張機能に関する一般情報については、次のセクションを参照して下さい。

- *セキュリティ状態*
- *モードと例外モデルに対するセキュリティ拡張機能の影響* P. B1-28
- *CP15 レジスタに対するセキュリティ拡張機能の影響* P. B3-71

B1.5.1 セキュリティ状態

セキュリティ拡張機能では、セキュア状態と非セキュア状態の 2 つのセキュリティ状態が定義されます。次のように、すべてのコードはセキュア状態と非セキュア状態のどちらかで実行されます。

- それぞれのセキュリティ状態は、個別の仮想メモリアドレス空間で動作します。
- システム制御の多くは、それぞれのセキュリティ状態について独自に設定できます。
- セキュリティ拡張機能が実装されていないシステムで使用可能なすべてのプロセッサモードは、それぞれのセキュリティ状態で使用することができます。

セキュリティ拡張機能では、非セキュア状態でのコード実行とセキュア状態でのコード実行との橋渡しをする追加のプロセッサモードである、モニタモードも定義されます。

次の特徴は、この 2 つのセキュリティ状態によって、通常のシステムで特権コードと非特権コードを分離することによって提供されるものよりも優れたセキュリティが提供されることを意味します。

- メモリシステムによって、セキュアとして割り当てられた物理メモリの領域に非セキュア状態からアクセスすることを防止する機構が提供されます。
- セキュア状態に適用されるシステム制御には、非セキュア状態からアクセスすることができません。
- 非セキュア状態からセキュア状態への遷移は、少数の例外によってのみ提供されます。
- セキュア状態から非セキュア状態への遷移は、少数の機構によってのみ提供されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The fundamental mechanism that determines the security state is the SCR.NS bit, see *c1, Secure Configuration Register (SCR)* on page B3-106:

- for all modes other than Monitor mode, the SCR.NS bit determines the security state for code execution
- code executing in Monitor mode is executed in the Secure state regardless of the value of the SCR.NS bit.

Code can change the SCR only if it is executing in the Secure state.

The general-purpose registers and the processor status registers are not banked between the Secure and the Non-secure states. When execution switches between the Non-secure and Secure security states, ARM expects that the values of these registers are switched by a kernel running mostly in Monitor mode.

Many of the system registers described in *Coprocessors and system control* on page B1-62 are banked between the Secure and Non-secure security states. A banked copy of a register applies only to execution in the appropriate security state. A small number of system registers are not banked but apply to both the Secure and Non-secure security states. Typically the registers that are not banked relate to global system configuration options that ARM expects to be common to the two security states.

Figure B1-2 on page B1-27 shows the normal transfers of control between different modes and security states.

Note

In Figure B1-2 on page B1-27, the route labelled as `MCR` is for an `MCR` instruction writing to the SCR, that sets SCR.NS to 1 (Non-secure) at a time when SCR.NS == 0 (Secure) and the processor is not in Monitor mode. This is a possible transfer, but ARM recommends that the value of SCR.NS is changed only by code executing in Monitor mode, see *Changing from Secure to Non-secure state* on page B1-27.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- オペレーティングシステム例外の多くは、セキュリティ状態を変更せずに処理することができます。

セキュリティ状態を決定する基本的な機構は、SCR.NS ビットです。*c1、セキュア構成レジスタ (SCR)* P. B3-106 を参照して下さい。

- モニタモード以外のすべてのモードで、コード実行のセキュリティ状態が SCR.NS ビットによって決定されます。
- モニタモードでのコード実行は、SCR.NS ビットの値に関係なく、セキュア状態で実行されます。

コードは、セキュア状態で実行されている場合にのみ SCR を変更することができます。

汎用レジスタとプロセッサステータス レジスタは、セキュア状態と非セキュア状態の間でバンクされません。ARM では、非セキュア状態とセキュア状態との間で実行が切り替わるときに、これらのレジスタの値が主にモニタモードで実行中のカーネルによって切り替えられることを前提としています。

コプロセッサとシステム制御 P. B1-62 に示すシステムレジスタの多くは、セキュアと非セキュアのセキュリティ状態との間でバンクされます。レジスタのバンクコピーは、適切なセキュリティ状態での実行にのみ適用されます。いくつかのシステムレジスタはバンクされず、セキュアと非セキュアの両方のセキュリティ状態に適用されます。通常、バンクされないレジスタは、2 つのセキュリティ状態に共通することが前提となっているグローバルなシステム構成オプションに関係します。

複数のモードとセキュリティ状態の間の通常の制御の遷移を、P. B1-27 の図 B1-2 に示します。

注

P. B1-27 の図 B1-2 で `MCR` というラベルが付いている経路は、SCR.NS == 0（セキュア）にセットされ、プロセッサがモニタモードでないときに、SCR.NS を 1（非セキュア）にセットする、SCR に対する `MCR` 命令による書き込みを表しています。この遷移は可能ですが、SCR.NS の値はモニタモードで実行中のコードでのみ変更することをお勧めします。*セキュア状態から非セキュア状態への変更* P. B1-27 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure B1-2 Security state, Monitor mode, and the SCR.NS bit

Note

It is important to distinguish between:

Monitor mode This is a processor mode that is only available when the Security Extensions are implemented. It is used in normal operation, as a mechanism to transfer between Secure and Non-secure state, as described in this section.

Monitor debug-mode

This is a debug mode and is available regardless of whether the Security Extensions are implemented. For more information, see *About the ARM Debug architecture* on page C1-3.

Changing from Secure to Non-secure state

The security state is controlled by the SCR.NS bit, and ARM recommends that the SCR is modified only in Monitor mode. Monitor mode is responsible for switching between Secure and Non-secure states.

To return to Non-secure state, set the SCR.NS bit to 1 and then perform an exception return.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

図 B1-2 セキュリティ状態、モニタモード、および SCR.NS ビット

注

次のモードの違いを認識しておくことが重要です。

モニタモード これは、セキュリティ拡張機能が実装されている場合にのみ使用可能になるプロセッサモードです。このセクションで説明されているように、セキュア状態と非セキュア状態との間を遷移するための機構として、通常の動作で使用されます。

モニタデバッグモード

これはデバッグモードで、セキュリティ拡張機能が実装されているかどうかに関係なく使用できます。詳細については、*ARM デバッグアーキテクチャについて* P. C1-3 を参照して下さい。

セキュア状態から非セキュア状態への変更

セキュリティ状態は、SCR.NS ビットによって制御されます。SCR はモニタモードでのみ変更することをお勧めします。モニタモードは、セキュア状態と非セキュア状態との切り替えに使用されます。

非セキュア状態に戻るには、SCR.NS ビットを 1 にセットしてから、例外からの復帰を実行します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

To avoid security holes, ARM strongly recommends that:

- you do not change from Secure to Non-secure state by using an `MSR` or `CPS` instruction to switch from Monitor mode to some other mode while SCR.NS is 1
- you do not use an `MCR` instruction that writes SCR.NS to change from Secure to Non-secure state. This means you should not alter the SCR.NS bit in any mode except Monitor mode.

The usual mechanism for changing from Secure to Non-secure state is an exception return.

Pseudocode details of Secure state operations

The `HaveSecurityExt()` function returns TRUE if the Security Extensions are implemented, and FALSE otherwise.

The following function returns TRUE if the Security Extensions are not implemented or the processor is in Secure state, and FALSE otherwise.

```
// IsSecure()
// ==========

boolean IsSecure()
    return !HaveSecurityExt() || SCR.NS == '0' || CPSR.M == '10110'; // Monitor mode
```

B1.5.2 Impact of the Security Extensions on the modes and exception model

This section summarizes the effect of the Security Extensions on the modes and exception model, to give a overview of the Security Extensions. When the Security Extensions are implemented:

- An additional mode, Monitor mode, is implemented. For more information, see *ARM processor modes* on page B1-6 and *Security states* on page B1-25.
- An additional exception, the *Secure Monitor Call* (SMC) exception, is implemented. This is generated by the `SMC` instruction. For more information, see *Secure Monitor Call (SMC) exception* on page B1-53 and *SMC (previously SMI)* on page B6-18.
- Because the SCTLR is banked between the Secure and Non-secure states, the V and VE bits are defined independently for the Secure and Non-secure states. For each state:
  - — the SCTLR.V bit controls whether the normal or the high exception vectors are used
  - — the SCTLR.VE bit controls whether the IRQ and FIQ vectors are IMPLEMENTATION DEFINED.

  For more information, see *Exception vectors and the exception base address* on page B1-30.
- The base address for the normal exception vectors is held in a CP15 register that is banked between the two security states. This register defines the base address used for exceptions handled in modes other than Monitor mode. Another CP15 register holds the base address for exceptions handled in Monitor mode. For more information, see *Exception vectors and the exception base address* on page B1-30.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

セキュリティホールを避けるために、次の対策を強くお勧めします。

- SCR.NS が 1 の間は、`MSR` 命令または `CPS` 命令を使用してモニタモードから他のモードに切り替えることによって、セキュア状態から非セキュア状態に切り替えないで下さい。
- `MCR` 命令を使用して SCR.NS に書き込むことによって、セキュア状態から非セキュア状態に変更しないで下さい。これは、モニタモード以外のモードでは、SCR.NS ビットを変更すべきではないことを意味します。

セキュア状態から非セキュア状態に変更するために一般に使用される機構は、例外からの復帰です。

セキュア状態動作の擬似コードの詳細

`HaveSecurityExt()` 関数は、セキュリティ拡張機能が実装されている場合に TRUE を返し、それ以外の場合は FALSE を返します。

次の関数は、セキュリティ拡張機能が実装されていない場合、またはプロセッサがセキュア状態の場合は TRUE を返し、それ以外の場合は FALSE を返します。

```
// IsSecure()
// ==========

boolean IsSecure()
    return !HaveSecurityExt() || SCR.NS == '0' || CPSR.M == '10110'; // Monitor mode
```

B1.5.2 モードと例外モデルに対するセキュリティ拡張機能の影響

このセクションでは、セキュリティ拡張機能の概要を示すために、モードと例外モデルに対するセキュリティ拡張機能の影響について説明します。セキュリティ拡張機能が実装されている場合は、次の規則が適用されます。

- 追加モードとして、モニタモードが実装されます。詳細については、*ARM プロセッサモード* P. B1-6 と *セキュリティ状態* P. B1-25 を参照して下さい。
- 追加の例外として、*セキュア モニタコール* (SMC) 例外が実装されます。この例外は、`SMC` 命令によって生成されます。詳細については、*セキュア モニタコール (SMC) 例外* P. B1-53 と *SMC (以前の SMI)* P. B6-18 を参照して下さい。
- SCTLR がセキュア状態と非セキュア状態との間でバンクされるため、V ビットと VE ビットがセキュア状態および非セキュア状態とは無関係に定義されます。状態によって次のような違いがあります。
  - — 通常の例外ベクタと上位例外ベクタのどちらが使用されるかが、SCTLR.V ビットによって制御されます。
  - — IRQ ベクタと FIQ ベクタが実装定義かどうかが、SCTLR.VE ビットによって制御されます。

  詳細については、*例外ベクタと例外ベースアドレス* P. B1-30 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- If an exception is taken in Monitor mode in Non-debug state, the SCR.NS bit is set to zero, see *c1, Secure Configuration Register (SCR)* on page B3-106. This forces Secure state entry for all exceptions. However, if an exception is taken in Monitor mode in Debug state, the SCR.NS bit is not set to zero.

  Note

  Many uses of the Security Extensions can be simplified if the system is designed so that exceptions cannot be taken in Monitor mode.

- Setting bits in the Secure Configuration Register causes one or more of external aborts, IRQs and FIQs to be handled in Monitor mode and to use the Monitor exception base address:
  — setting the SCR.EA bit to 1 means external aborts are handled in Monitor mode, instead of Abort mode
  — setting the SCR.FIQ bit to 1 means FIQs are handled in Monitor mode, instead of FIQ mode
  — setting the SCR.IRQ bit to 1 means IRQs are handled in Monitor mode, instead of IRQ mode.

  For more information see:
  — *Control of exception handling by the Security Extensions* on page B1-41
  — *c1, Secure Configuration Register (SCR)* on page B3-106.

- Setting bits in the Secure Configuration Register prevents code executing in Non-secure state from being able to mask one or both of asynchronous aborts and FIQs:
  — Setting the SCR.AW bit to 1 prevents Non-secure setting of CPSR.A to 1.
  — Setting the SCR.FW bit to 1 prevents Non-secure setting of CPSR.F to 1. For details of how this setting interacts with NMFIs see *Non-maskable fast interrupts* on page B1-18.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- 通常の例外ベクタのベースアドレスは、2 つのセキュリティ状態の間でバンクされる CP15 レジスタに保持されます。このレジスタによって、モニタモード以外のモードで処理される例外で使用されるベースアドレスが定義されます。もう 1 つの CP15 レジスタには、モニタモードで処理される例外のベースアドレスが保持されます。詳細については、*例外ベクタと例外ベースアドレス* P. B1-30 を参照して下さい。

- 非デバッグ状態のモニタモードで例外が取得された場合は、SCR.NS ビットが 0 に設定されます。*c1、セキュア構成レジスタ (SCR)* P. B3-106 を参照して下さい。これによって、すべての例外に対して強制的にセキュア状態への遷移が行われます。ただし、デバッグ状態のモニタモードで例外が取得された場合は、SCR.NS ビットが 0 にセットされません。

  注

  モニタモードでは例外を取得できないようにシステムが設計されていれば、セキュリティ拡張機能の多くの用途を単純化することができます。

- セキュア構成レジスタ内のビットをセットすると、1 つ以上の外部アボート、IRQ、FIQ がモニタモードで処理され、モニタ例外ベースアドレスが使用されます。
  – SCR.EA ビットを 1 にセットすると、外部アボートがアボートモードではなく、モニタモードで処理されます。
  – SCR.FIQ ビットを 1 にセットすると、FIQ が FIQ モードではなく、モニタモードで処理されます。
  – SCR.IRQ ビットを 1 にセットすると、IRQ が IRQ モードではなく、モニタモードで処理されます。

  詳細については、次のセクションを参照して下さい。
  – *セキュリティ拡張機能による例外処理の制御* P. B1-41
  – *c1、セキュア構成レジスタ (SCR)* P. B3-106

- セキュア構成レジスタ内のビットをセットすることによって、非セキュア状態でのコード実行で非同期アボートと FIQ のどちらかまたは両方をマスクできなくなります。
  – SCR.AW ビットを 1 にセットすることによって、CPSR.A を 1 にする非セキュア設定が防止されます。
  – SCR.FW ビットを 1 にセットすることによって、CPSR.F を 1 にする非セキュア設定が防止されます。この設定が NMFI にどのように影響するかについては、*マスク不能高速割り込み* P. B1-18 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.6 Exceptions

An exception causes the processor to suspend program execution to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. Exceptions can be generated by internal and external sources.

Normally, when an exception is taken the processor state is preserved immediately, before handling the exception. This means that, when the event has been handled, the original state can be restored and program execution resumed from the point where the exception was taken.

More than one exception might be generated at the same time, and a new exception can be generated while the processor is handling an exception.

The following sections describe exception handling in general:

- *Exception vectors and the exception base address*
- *Exception priority order* on page B1-33
- *Exception entry* on page B1-34
- *Exception return* on page B1-38
- *Exception-handling instructions* on page B1-41
- *Control of exception handling by the Security Extensions* on page B1-41
- *Low interrupt latency configuration* on page B1-43.
- *Wait For Event and Send Event* on page B1-44
- *Wait For Interrupt* on page B1-47.

The following sections give details of each exception:

- *Reset* on page B1-48
- *Undefined Instruction exception* on page B1-49
- *Supervisor Call (SVC) exception* on page B1-52
- *Secure Monitor Call (SMC) exception* on page B1-53
- *Prefetch Abort exception* on page B1-54
- *Data Abort exception* on page B1-55
- *IRQ exception* on page B1-58
- *FIQ exception* on page B1-60.

B1.6.1 Exception vectors and the exception base address

When an exception is taken, processor execution is forced to an address that corresponds to the type of exception. These addresses are called the *exception vectors*.

By default, the exception vectors are eight consecutive word-aligned memory addresses, starting at an *exception base address*. Table B1-3 on page B1-31 shows the assignment of the exceptions to the eight memory addresses.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B1.6 例外

例外が発生すると、プロセッサは、プログラムの実行を中断して、外部で生成された割り込みや未定義命令の実行の試みなどのイベントを処理します。例外は内部ソースと外部ソースによって生成される可能性があります。

通常は、例外が取得されたときに、それを処理する直前のプロセッサ状態が保存されます。これは、イベントの処理が完了したときに、元の状態を復元し、プログラムの実行を例外が取得された時点から再開できることを意味します。

複数の例外が同時に生成されたり、プロセッサが例外を処理している間に別の例外が生成されたりする可能性があります。

一般的な例外処理については、次に示すセクションで説明します。

- *例外ベクタと例外ベースアドレス*
- *例外の優先順位* P. B1-33
- *例外の開始* P. B1-34
- *例外からの復帰* P. B1-38
- *例外処理命令* P. B1-41
- *セキュリティ拡張機能による例外処理の制御* P. B1-41
- *低割り込みレイテンシ構成* P. B1-43
- *イベント待ちとイベント送信* P. B1-44
- *割り込み待ち* P. B1-47

それぞれの例外の詳細については、次に示すセクションで説明します。

- *リセット* P. B1-48
- *未定義命令例外* P. B1-49
- *スーパバイザコール (SVC) 例外* P. B1-52
- *セキュア モニタコール (SMC) 例外* P. B1-53
- *プリフェッチアボート例外* P. B1-54
- *データアボート例外* P. B1-55
- *IRQ 例外* P. B1-58
- *FIQ 例外* P. B1-60

B1.6.1 例外ベクタと例外ベースアドレス

例外が取得されると、プロセッサの実行が例外のタイプに対応したアドレスに移行されます。このアドレスは、*例外ベクタ*と呼ばれます。

デフォルトで、例外ベクタは、*例外ベースアドレス*で始まる、8 つの連続するワードアラインのメモリアドレスです。8 つのメモリアドレスに割り当てられた例外を、P. B1-31 の表 B1-3 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B1-3 Offsets from exception base addresses

Exception offset	Exception that is vectored at that offset from:	
	Monitor exception base address[a]	Base address for all other exceptions
0x00	Not used	Reset
0x04	Not used	Undefined Instruction
0x08	Secure Monitor Call (SMC)	Supervisor Call (SVC)
0x0C	Prefetch Abort	Prefetch Abort
0x10	Data Abort	Data Abort
0x14	Not used	Not used
0x18	IRQ (interrupt)	IRQ (interrupt)
0x1C	FIQ (fast interrupt)	FIQ (fast interrupt)

a. This column applies only if the Security Extensions are implemented.

The default exception vectors for the IRQ and FIQ exceptions can be changed by setting the SCTLR.VE bit to 1, as described in *Vectored interrupt support* on page B1-32.

If the Security Extensions are not implemented there is a single exception base address. This is controlled by the SCTLR.V bit:

V == 0 Exception base address = `0x00000000`. This setting is referred to as normal vectors, or as low vectors.

V == 1 Exception base address = `0xFFFF0000`. This setting is referred to as high vectors, or *Hivecs*.

Note

Use of the Hivecs setting, V == 1, is deprecated in ARMv7-R. ARM recommends that Hivecs is used only in ARMv7-A implementations.

If the Security Extensions are implemented there are three exception base addresses:

- the Non-secure exception base address is used for all exceptions that are processed in Non-secure state
- the Secure exception base address is used for all exceptions that are processed in Secure state but not in Monitor mode
- the Monitor exception base address is used for all exceptions that are processed in Monitor mode.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B1-3 例外ベースアドレスからのオフセット

例外オフセット	このオフセットのベクタに対応する例外	
	モニタ例外ベースアドレス[a]	その他のすべての例外用ベースアドレス
0x00	未使用	リセット
0x04	未使用	未定義命令
0x08	セキュア モニタコール (SMC)	スーパバイザコール (SVC)
0x0C	プリフェッチアボート	プリフェッチアボート
0x10	データアボート	データアボート
0x14	未使用	未使用
0x18	IRQ（割り込み）	IRQ（割り込み）
0x1C	FIQ（高速割り込み）	FIQ（高速割り込み）

a. この列は、セキュリティ拡張機能が実装されている場合にのみ適用されます。

ベクタ割り込みのサポート P. B1-32 で説明されているように、IRQ 例外と FIQ 例外のデフォルトの例外ベクタは、SCTLR.VE ビットを 1 にセットして変更できます。

セキュリティ拡張機能が実装されていない場合は、例外ベースアドレスが 1 つしか存在しません。このアドレスは、SCTLR.V ビットによって次のように制御されます。

V == 0 例外ベースアドレス = `0x00000000`。この設定は、通常ベクタまたはローベクタと呼ばれます。

V == 1 例外ベースアドレス = `0xFFFF0000`。この設定は、ハイベクタまたは *Hivecs* と呼ばれます。

注

ARMv7-R では、Hivecs 設定を示す V == 1 の使用は推奨されません。Hivecs は ARMv7-A 実装でのみ使用することをお勧めします。

セキュリティ拡張機能が実装されている場合は、次の 3 つの例外ベースアドレスが存在します。

- 非セキュア例外ベースアドレスは、非セキュア状態で処理されるすべての例外に対して使用されます。
- セキュア例外ベースアドレスは、モニタモード以外のセキュア状態で処理されるすべての例外に対して使用されます。
- モニタ例外ベースアドレスは、モニタモードで処理されるすべての例外に対して使用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

See *CPSR M field and A, I, and F mask bit values on exception entry* on page B1-36 to determine the mode in which an exception is processed. If that mode is Monitor mode then the exception is processed in Secure state, otherwise the exception is processed in the current security state, determined at the time when the exception is taken.

The Non-secure exception base address is controlled by the SCTLR.V bit in the Non-secure SCTLR:

V == 0 The exception base address is the value of the Non-secure *Vector Base Address Register* (VBAR), see *c12, Vector Base Address Register (VBAR)* on page B3-148.

V == 1 Exception base address = `0xFFFF0000`. This setting is often referred to as *Hivecs*.

The Secure exception base address is controlled similarly, by the Secure SCTLR.V bit and the Secure VBAR.

The Monitor exception base address is always the value of the *Monitor Vector Base Address Register* (MVBAR), see *c12, Monitor Vector Base Address Register (MVBAR)* on page B3-149.

Vectored interrupt support

By default, the IRQ and FIQ exception vectors are at fixed offsets from the exception base address that is being used. This is consistent with previous versions of the ARM architecture. With this default configuration, each of the FIQ and IRQ handlers typically starts with an instruction sequence that determines the cause of the interrupt and then branches to an appropriate routine to handle it.

Support for vectored interrupts means an interrupt controller can prioritize interrupts and provide the address of the required interrupt handler directly to the processor, for use as the interrupt vector. Vectored interrupt behavior is enabled by setting the SCTLR.VE bit to 1, see:

- *c1, System Control Register (SCTLR)* on page B3-96 for a VMSA implementation
- *c1, System Control Register (SCTLR)* on page B4-45 for a PMSA implementation.

The hardware that supports vectored interrupts is IMPLEMENTATION DEFINED.

For backwards compatibility, the vectored interrupt mechanism is disabled on reset.

When the Security Extensions are implemented:

- The SCTLR.VE bit is banked between Secure and Non-secure states to provide independent control of whether vectored interrupt support is enabled.
- Interrupts can be trapped to Monitor mode, by setting either or both of the SCR.IRQ and SCR.FIQ bits to 1. When an interrupt is trapped to Monitor mode it uses the vector in the vector table addressed by the Monitor exception base address held in MVBAR, regardless of the value of either banked copy of the SCTLR.VE bit.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

例外が処理されるモードを特定するには、*例外の開始時点の CPSR M フィールドと、A、I、F の各マスクビットの値* P. B1-36 を参照して下さい。そのモードがモニタモードの場合は、例外がセキュア状態で処理されますが、それ以外の場合は、例外が取得された時点で決定される現在のセキュリティ状態で例外が処理されます。

非セキュア例外ベースアドレスは、非セキュア SCTLR 内の SCTLR.V ビットによって制御されます。

V == 0 例外ベースアドレスは、非セキュア *ベクタ ベースアドレスレジスタ*(VBAR) の値です。*c12、ベクタ ベースアドレスレジスタ (VBAR)* P. B3-148 を参照して下さい。

V == 1 例外ベースアドレス = `0xFFFF0000`。多くの場合、この設定は *Hivecs* と呼ばれます。

同様に、セキュア例外ベースアドレスは、セキュア SCTLR.V ビットとセキュア VBAR によって制御されます。

モニタ例外ベースアドレスは、常に*モニタベクタ ベースアドレスレジスタ*(MVBAR) の値です。*c12、モニタベクタ ベースアドレス レジスタ (MVBAR)* P. B3-149 を参照して下さい。

ベクタ割り込みのサポート

デフォルトで、IRQ 例外ベクタと FIQ 例外ベクタは、使用される例外ベースアドレスから固定オフセットの位置に存在します。これは、以前のバージョンの ARM アーキテクチャから一貫しています。このデフォルトの構成を使用して、FIQ ハンドラと IRQ ハンドラは通常、割り込みの原因を特定する命令シーケンスを開始して、その割り込みを処理するためのルーチンに分岐します。

ベクタ割り込みのサポートは、割り込みコントローラが割り込みの優先順位を付け、割り込みベクタとして使用するために必要な割り込みハンドラのアドレスを直接プロセッサに渡すことができることを意味します。ベクタ割り込みの動作は、SCTLR.VE ビットを 1 に設定することによって有効になります。次のセクションを参照して下さい。

- *c1、システム制御レジスタ (SCTLR)* P. B3-96（VMSA 実装の場合）
- *c1、システム制御レジスタ (SCTLR)* P. B4-45 （PMSA 実装の場合）

ベクタ割り込みをサポートするハードウェアは、実装定義です。

下位互換性を維持するために、ベクタ割り込みの機構はリセットで無効になります。

セキュリティ拡張機能が実装されている場合、次の規則が適用されます。

- SCTLR.VE ビットがセキュア状態と非セキュア状態との間でバンクされ、ベクタ割り込みのサポートが有効かどうかに関係しない制御が提供されます。
- 割り込みは、SCR.IRQ ビットと SCR.FIQ ビットのどちらかまたは両方を 1 にセットすることによって、モニタモードにトラップすることができます。割り込みがモニタモードにトラップされた場合は、SCTLR.VE ビットのバンクコピーの値に関係なく、MVBAR に保存されたモニタ例外ベースアドレスによって指定されるベクタテーブル内のベクタが使用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Operation

In pseudocode, the current exception base address for exceptions processed in Monitor mode is determined by reading MVBAR, and for other exceptions by the following function:

```
// ExcVectorBase()
// ===============

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then  // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    elsif HaveSecurityExt() then
        return VBAR;
    else
        return Zeros(32);
```

B1.6.2 Exception priority order

In principle a number of different synchronous exceptions can be generated by a single instruction. The following principles determine which synchronous exception is taken:

- No instruction is valid if it has a synchronous Prefetch Abort exception associated with it. Therefore, other synchronous exceptions are not taken in this case.
- An instruction that generates an Undefined Instruction exception cannot cause any memory access, and therefore cannot cause a Data Abort exception.
- All other synchronous exceptions are mutually exclusive and are derived from a decode of the instruction.

The ARM architecture does not define when asynchronous exceptions are taken. Therefore the prioritization of asynchronous exceptions relative to other exceptions, both synchronous and asynchronous, depends on the implementation.

The CPSR includes a mask bit for each type of asynchronous exception. Setting one of these bits to 1 prevents the corresponding asynchronous exception from being taken. Taking an exception sets an exception-dependent subset of these mask bits.

Note

- The subset of the CPSR mask bits that is set on taking an exception prioritizes the execution of FIQ handlers over that of IRQ and asynchronous abort handlers.
- A special requirement applies to asynchronous watchpoints - see *Debug event prioritization* on page C3-43.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

動作

擬似コードでは、モニタモードで処理される例外の現在の例外ベースアドレスが MVBAR を読み出すことによって決定され、その他の例外の現在の例外ベースアドレスが次の関数によって決定されます。

```
// ExcVectorBase()
// ===============

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then  // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    elsif HaveSecurityExt() then
        return VBAR;
    else
        return Zeros(32);
```

B1.6.2 例外の優先順位

原則として、単一の命令で複数の異なる同期例外を生成することができます。次の原則によって、どの同期例外が取得されるかが決定されます。

- 同期プリフェッチアボート例外が関連づけられた命令は無効です。したがって、この場合、その他の同期例外は取得されません。
- 未定義命令例外を生成する命令は、メモリアクセスを引き起こすことができないため、データアボート例外を引き起こすことができません。
- その他のすべての同期例外は相互排他的であり、命令のデコードから抽出されます。

ARM アーキテクチャでは、非同期例外が取得されるタイミングが定義されていません。そのため、その他の例外（同期と非同期を含む）に対する非同期例外の優先度は、実装に依存します。

CPSR には、非同期例外のタイプごとのマスクビットが含まれています。これらのビットのいずれかを 1 にセットすると、対応する非同期例外が取得されなくなります。例外が取得されると、これらのマスクビットの例外依存サブセットがセットされます。

注

- 例外の取得時にセットされる CPSR マスクビットのサブセットでは、IRQ ハンドラや非同期アボートハンドラの実行よりも FIQ ハンドラの実行が優先されます。
- 非同期ウォッチポイントには、特別な要件が適用されます。*デバッグイベントの優先度付け* P. C3-43 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.6.3 Exception entry

On taking an exception:

1. The value of the CPSR is saved in the SPSR for the exception mode that is handling the exception.
2. The value of (PC + exception-dependent offset) is saved in the LR for the exception mode that is handling the exception, see Table B1-4.
3. The CPSR and PC are updated with information for the exception handler:
   - The CPSR is updated with new context information. This includes:
     - — Setting CPSR.M to the processor mode in which the exception is to be handled.
     - — Disabling appropriate classes of interrupt, to prevent uncontrolled nesting of exception handlers. For more information, see Table B1-6 on page B1-36, Table B1-7 on page B1-37, and Table B1-8 on page B1-37.
     - — Setting the instruction set state to the instruction set chosen for exception entry, see *Instruction set state on exception entry* on page B1-35.
     - — Setting the endianness to the value chosen for exception entry, see *CPSR.E bit value on exception entry* on page B1-38.
     - — Clearing the IT[7:0] bits to 0.

     For more information, see *CPSR M field and A, I, and F mask bit values on exception entry* on page B1-36.
   - The appropriate exception vector is loaded to the PC, see *Exception vectors and the exception base address* on page B1-30.
4. Execution continues from the address held in the PC.

At step 2 of the exception entry, the address saved in the LR depends on:
- the Exception type
- the instruction set state in which the processor is executing when the exception occurs.

Table B1-4 shows the LR value saved for all cases:

Table B1-4 Link Register value saved on exception entry

Exception	Base LR value [a]	Offset, for processor state of:[a]		
		ARM	Thumb or ThumbEE	Jazelle
Reset	UNKNOWN	-	-	-
Undefined Instruction	Address of the undefined instruction	+ 4	+2	+2 or +4[b]
SVC	Address of SVC instruction	+ 4	+2	-[c]
SMC	Address of SMC instruction	+ 4	+4	-[c]

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B1.6.3 例外の開始

例外を取得すると、次の処理が実行されます。

1. CPSR の値が、例外を処理する例外モード用の SPSR に保存されます。
2. （PC + 例外依存オフセット）の値が、例外を処理する例外モード用の LR に保存されます。表 B1-4 を参照して下さい。
3. CPSR と PC が、例外ハンドラの情報に更新されます。
   - CPSR が新しいコンテキスト情報に更新されます。これには、次のものが含まれます。
     - – CPSR.M が、例外が処理されるプロセッサモードに設定されます。
     - – 例外ハンドラが無制御にネストすることを避けるため、該当する割り込みクラスが禁止されます。詳細については、P. B1-36 の表 B1-6、P. B1-37 の表 B1-7、P. B1-37 の表 B1-8 を参照して下さい。
     - – 例外の開始用に選択された命令セットへ、命令セット状態が設定されます。*例外の開始時点の命令セット状態* P. B1-35 を参照して下さい。
     - – 例外の開始用に選択された値に、エンディアン形式が設定されます。*例外の開始時点の CPSR.E ビットの値* P. B1-38 を参照して下さい。
     - – IT[7:0] ビットが 0 にクリアされます。

     詳細については、*例外の開始時点の CPSR M フィールドと、A、I、F の各マスクビットの値* P. B1-36 を参照して下さい。
   - 該当する例外ベクタが PC にロードされます。*例外ベクタと例外ベースアドレス* P. B1-30 を参照して下さい。
4. PC に保持されているアドレスから、実行が継続されます。

例外の開始の手順 2 で LR に保存されるアドレスは、次の要素に依存します。
- 例外タイプ
- 例外発生時にプロセッサが実行していた命令セット状態

すべてのケースで保存される LR 値を、表 B1-4 に示します。

表 B1-4 例外の開始時点で保存されるリンクレジスタの値

例外	ベース LR の値 [a]	プロセッサ状態に対応するオフセット [a]		
		ARM	Thumb または ThumbEE	Jazelle
リセット	未知	–	–	–
未定義命令	未定義命令のアドレス	+ 4	+2	+ 2 または + 4[b]
SVC	SVC 命令のアドレス	+ 4	+2	–[c]
SMC	SMC 命令のアドレス	+ 4	+4	–[c]

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B1-4 Link Register value saved on exception entry (continued)

Exception	Base LR value [a]	Offset, for processor state of:[a]		
		ARM	Thumb or ThumbEE	Jazelle
Prefetch Abort	Address of aborted instruction fetch	+ 4	+4	+ 4
Data Abort	Address of instruction that generated the abort	+ 8	+8	+ 8
IRQ or FIQ	Address of next instruction to execute	+ 4	+4	+ 4

a. Except for the Reset exception, the value saved in the LR is the base LR value plus the offset value for the processor state immediately before the exception entry.
b. In Jazelle state, Undefined Instruction exceptions can only happen on a processor that includes a trivial implementation of Jazelle state. On such a processor, if an exception return instruction writes {CPSR.J, CPSR.T} to 0b10, the processor takes an Undefined Instruction exception when it next attempts to execute an instruction. It is IMPLEMENTATION DEFINED whether the processor uses an offset of +2 or +4 in these circumstances, but it must always use the same offset.
c. SVC and SMC exceptions cannot occur in Jazelle state.

Instruction set state on exception entry

Exception handlers always execute in either Thumb state or ARM state. Which state they execute in is determined by the Thumb Exception enable bit, SCTLR.TE, see:

- *c1, System Control Register (SCTLR)* on page B3-96 for a VMSA implementation
- *c1, System Control Register (SCTLR)* on page B4-45 for a PMSA implementation.

On exception entry, the CPSR.T and CPSR.J bits are set to values that depend on the SCTLR.TE value, as shown in Table B1-5:

Table B1-5 CPSR.J and CPSR.T bit values on exception entry

SCTLR.TE	CPSR.J	CPSR.T	Exception handler state
0	0	0	ARM
1	0	1	Thumb

When the Security Extensions are implemented, the SCTLR is banked for Secure and Non-secure states, and therefore the TE bit value might be different for Secure and Non-secure states. The SCTLR.TE bit for the security state in which the exception is handled determines the instruction set state for the exception handler. This means the exception handlers might run in different instruction set states, depending on the security state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B1-4 例外の開始時点で保存されるリンクレジスタの値（続き）

例外	ベース LR の値 [a]	プロセッサ状態に対応するオフセット [a]		
		ARM	Thumb または ThumbEE	Jazelle
プリフェッチアボート	アボートされた命令フェッチのアドレス	+ 4	+4	+ 4
データアボート	アボートを生成した命令のアドレス	+ 8	+8	+ 8
IRQ または FIQ	次に実行される命令のアドレス	+ 4	+4	+ 4

a. リセット例外を除いて、LR に保存される値は、ベース LR の値に、例外の開始直前のプロセッサ状態に対応するオフセット値を加えた値になります。
b. Jazelle 状態での未定義命令例外は、Jazelle 状態のトリビアル実装を含むプロセッサでのみ発生します。このようなプロセッサで、例外からの復帰命令によって {CPSR.J, CPSR.T} に 0b10 が書き込まれた場合は、次の命令の実行が試みられたときに、プロセッサにより未定義命令例外が取得されます。このような状況でプロセッサが + 2 と + 4 のどちらのオフセットを使用するかは実装定義ですが、常に同じオフセットを使用する必要があります。
c. SVC 例外と SMC 例外が Jazelle 状態で発生することはありません。

例外の開始時点の命令セット状態

例外ハンドラは、常に Thumb 状態または ARM 状態のどちらかで実行されます。どちらの状態で実行されるかは、Thumb 例外イネーブルビットである SCTLR.TE によって決定されます。次のセクションを参照して下さい。

- *c1、システム制御レジスタ (SCTLR)* P. B3-96（VMSA 実装の場合）
- *c1、システム制御レジスタ (SCTLR)* P. B4-45 （PMSA 実装の場合）

表 B1-5 に示すように、例外の開始時点で、CPSR.T ビットと CPSR.J ビットは、SCTLR.TE の値に依存する値にセットされます。

表 B1-5 例外の開始時点の CPSR.J ビットと CPSR.T ビットの値

			例外ハンドラの状態
0	0	0	ARM
1	0	1	Thumb

セキュリティ拡張機能が実装されている場合は、セキュア状態と非セキュア状態で SCTLR がバンクされるため、TE ビットの値がセキュア状態と非セキュア状態で異なる可能性があります。例外が処理されるセキュリティ状態の SCTLR.TE ビットによって、例外ハンドラの命令セット状態が決定されます。これは、セキュリティ状態によって、例外ハンドラが異なる命令セット状態で実行される可能性があることを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CPSR M field and A, I, and F mask bit values on exception entry

On exception entry, the processor mode is set to one of the exception modes and the CPSR[A,I,F] interrupt disable (mask) bits are set to new values:

- the CPSR.I bit is always set to 1, to disable IRQs
- the CPSR.M (mode), CPSR.A (asynchronous abort disable), and CPSR.F (FIQ disable) bits are set to values that depend:
  - on the exception type
  - if the Security Extensions are implemented, on the security state and some bits of the SCR, see *c1, Secure Configuration Register (SCR)* on page B3-106.

The new values are shown in:

- Table B1-6, for an implementation that does not include the Security Extensions
- Table B1-7 on page B1-37, for an implementation that includes the Security Extensions, when the security state is Secure (NS == 0).
- Table B1-8 on page B1-37, for an implementation that includes the Security Extensions, when the security state is Non-secure (NS == 1).

In these tables, Unchanged indicates that the bit value is unchanged from its value when the exception was taken.

Table B1-6 A and F bit values on exception entry, without Security Extensions

Exception	Exception mode	CPSR.A	CPSR.F
Reset	Supervisor	1	1
Undefined Instruction	Undefined	Unchanged	Unchanged
Supervisor Call (SVC)	Supervisor	Unchanged	Unchanged
All aborts	Abort	1	Unchanged
IRQ	IRQ	1	Unchanged
FIQ	FIQ	1	1

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

例外の開始時点の CPSR M フィールドと、A、I、F の各マスクビットの値

例外の開始時点で、プロセッサモードが例外モードのいずれかに設定され、CPSR[A、I、F] 割り込み禁止（マスク）ビットが新しい値に設定されます。

- CPSR.I ビットは必ず 1 に設定され、IRQ が禁止されます。
- CPSR.M（モード）、CPSR.A（非同期アボート禁止）、CPSR.F（FIQ 禁止）の各ビットは、次の要因に応じた値に設定されます。
  - 例外タイプ
  - セキュリティ拡張機能が実装されている場合は、セキュリティ状態と SCR の一部のビット（*c1、セキュア構成レジスタ (SCR)* P. B3-106 を参照）

新しい値については、次の表を参照して下さい。

- 表 B1-6（セキュリティ拡張機能を含まない実装の場合）
- P. B1-37 の表 B1-7（セキュリティ拡張機能を含む実装で、セキュリティ状態がセキュア（NS == 0）の場合）
- P. B1-37 の表 B1-8（セキュリティ拡張機能を含む実装で、セキュリティ状態が非セキュア（NS == 1）の場合）

これらの表の「変更なし」は、ビットの値が、例外が取得された時点の値から変更されないことを意味します。

表 B1-6 セキュリティ拡張機能が実装されていない場合の例外の開始時点の A ビットと F ビットの値

例外	例外モード	CPSR.A	CPSR.F
リセット	スーパバイザ	1	1
未定義命令	未定義	変更なし	変更なし
スーパバイザコール (SVC)	スーパバイザ	変更なし	変更なし
すべてのアボート	アボート	1	変更なし
IRQ	IRQ	1	変更なし
FIQ	FIQ	1	1

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B1-7 A and F bit values on exception entry, with Security Extensions and NS == 0

Exception	SCR bits			Exception mode	NS == 0, Secure	
	EA	IRQ	FIQ		CPSR.A	CPSR.F
Reset	x	x	x	Supervisor	1	1
Undefined Instruction	x	x	x	Undefined	Unchanged	Unchanged
Supervisor Call (SVC)	x	x	x	Supervisor	Unchanged	Unchanged
Secure Monitor Call (SMC)	x	x	x	Monitor	1	1
All external aborts	0	x	x	Abort	1	Unchanged
	1	x	x	Monitor	1	1
All internal aborts	x	x	x	Abort	1	Unchanged
IRQ	x	0	x	IRQ	1	Unchanged
	x	1	x	Monitor	1	1
FIQ	x	x	0	FIQ	1	1
	x	x	1	Monitor	1	1

Table B1-8 A and F bit values on exception entry, with Security Extensions and NS == 1

Exception	SCR bits					Exception mode	NS == 1, Non-secure	
	EA	IRQ	FIQ	AW	FW		CPSR.A	CPSR.F
Reset	x	x	x	x	x	Supervisor	1	1
Undefined Instruction	x	x	x	x	x	Undefined	Unchanged	Unchanged
Supervisor Call (SVC)	x	x	x	x	x	Supervisor	Unchanged	Unchanged
Secure Monitor Call (SMC)	x	x	x	x	x	Monitor	1	1
All external aborts	0	x	x	0	x	Abort	Unchanged	Unchanged
	0	x	x	1	x	Abort	1	Unchanged
	1	x	x	x	x	Monitor	1	1

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B1-7 セキュリティ拡張機能が実装され、NS = 0 に設定されている場合の例外の開始時点の A ビットと F ビットの値

例外	SCR ビット			例外モード	NS = 0、セキュア	
	EA	IRQ	FIQ		CPSR.A	CPSR.F
リセット	x	x	x	スーパバイザ	1	1
未定義命令	x	x	x	未定義	変更なし	変更なし
スーパバイザコール (SVC)	x	x	x	スーパバイザ	変更なし	変更なし
セキュア モニタコール (SMC)	x	x	x	モニタ	1	1
すべての外部アボート	0	x	x	アボート	1	変更なし
	1	x	x	モニタ	1	1
すべての内部アボート	x	x	x	アボート	1	変更なし
IRQ	x	0	x	IRQ	1	変更なし
	x	1	x	モニタ	1	1
FIQ	x	x	0	FIQ	1	1
	x	x	1	モニタ	1	1

表 B1-8 セキュリティ拡張機能が実装され、NS = 1 に設定されている場合の例外の開始時点の A ビットと F ビットの値

例外	SCR ビット					例外モード	NS = 1、非セキュア	
	EA	IRQ	FIQ	AW	FW		CPSR.A	CPSR.F
リセット	x	x	x	x	x	スーパバイザ	1	1
未定義命令	x	x	x	x	x	未定義	変更なし	変更なし
スーパバイザコール (SVC)	x	x	x	x	x	スーパバイザ	変更なし	変更なし
セキュア モニタコール (SMC)	x	x	x	x	x	モニタ	1	1
すべての外部アボート	0	x	x	0	x	アボート	変更なし	変更なし
	0	x	x	1	x	アボート	1	変更なし
	1	x	x	x	x	モニタ	1	1

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B1-8 A and F bit values on exception entry, with Security Extensions and NS == 1 (continued)

Exception	SCR bits					Exception mode	NS == 1, Non-secure	
	EA	IRQ	FIQ	AW	FW		CPSR.A	CPSR.F
All internal aborts	x	x	x	0	x	Abort	Unchanged	Unchanged
	x	x	x	1	x	Abort	1	Unchanged
IRQ	x	0	x	0	x	IRQ	Unchanged	Unchanged
	x	0	x	1	x	IRQ	1	Unchanged
	x	1	x	x	x	Monitor	1	1
FIQ	x	x	0	0	0	FIQ	Unchanged	Unchanged
	x	x	0	0	1	FIQ	Unchanged	1
	x	x	0	1	0	FIQ	1	Unchanged
	x	x	0	1	1	FIQ	1	1
	x	x	1	x	x	Monitor	1	1

CPSR.E bit value on exception entry

On exception entry, the CPSR.E bit is set to the value of the SCTLR.EE bit. This bit of the CPSR controls the load and store endianness for data handling by the exception handler, see the bit description in *Format of the CPSR and SPSRs* on page B1-16. For the description of the SCTLR see:

- *c1, System Control Register (SCTLR)* on page B3-96 for a VMSA implementation
- *c1, System Control Register (SCTLR)* on page B4-45 for a PMSA implementation.

B1.6.4 Exception return

In the ARM architecture, *exception return* requires the simultaneous restoration of the PC and CPSR to values that are consistent with the desired state of execution on returning from the exception. Normally, this is the state of execution just before the exception was taken, but it can be different in some circumstances, for example if the exception handler performed instruction emulation.

Typically, this involves returning to one of:

- the instruction boundary at which an asynchronous exception was taken
- the instruction following an `SVC` or `SMC` instruction, for an exception generated by one of those instructions
- the instruction that caused the exception, after the reason for the exception has been removed

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B1-8 セキュリティ拡張機能が実装され、NS = 1 に設定されている場合の例外の開始時点の A ビットと F ビットの値（続

例外	SCR ビット					例外モード	NS = 1、非セキュア	
	EA	IRQ	FIQ	AW	FW		CPSR.A	CPSR.F
すべての内部アボート	x	x	x	0	x	アボート	変更なし	変更なし
	x	x	x	1	x	アボート	1	変更なし
IRQ	x	0	x	0	x	IRQ	変更なし	変更なし
	x	0	x	1	x	IRQ	1	変更なし
	x	1	x	x	x	モニタ	1	1
FIQ	x	x	0	0	0	FIQ	変更なし	変更なし
	x	x	0	0	1	FIQ	変更なし	1
	x	x	0	1	0	FIQ	1	変更なし
	x	x	0	1	1	FIQ	1	1
	x	x	1	x	x	モニタ	1	1

例外の開始時点の CPSR.E ビットの値

例外の開始時点に、CPSR.E ビットは SCTLR.EE ビットの値に設定されます。CPSR のこのビットによって、例外ハンドラによるデータ処理用のロードとストアのエンディアン形式が制御されます。*CPSR と SPSR のフォーマット* P. B1-16 のビットの説明を参照して下さい。SCTLR については、次のセクションを参照して下さい。

- *c1、システム制御レジスタ (SCTLR)* P. B3-96（VMSA 実装の場合）
- *c1、システム制御レジスタ (SCTLR)* P. B4-45 （PMSA 実装の場合）

B1.6.4 例外からの復帰

ARM アーキテクチャでは、*例外からの復帰*によって、PC と CPSR が同時に、例外からの復帰時点で必要な実行状態と矛盾しない値に復元される必要があります。通常、これは例外が取得される直前の実行状態ですが、例外ハンドラが命令エミュレーションを実行する場合などの一部の状況では異なる可能性があります。

一般的に、これには次のいずれかへの復帰が含まれます。

- 非同期例外が取得された命令境界
- `SVC` 命令または `SMC` 命令によって生成された例外の場合は、これらの命令に続く命令
- 例外の原因が取り除かれた後で、例外を引き起こした命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- the subsequent instruction, if the instruction that caused the exception has been emulated in the exception handler.

The ARM architecture makes no requirement that exception return must be to any particular place in the execution stream. However, the architecture does have a *preferred exception return* for each exception other than Reset. The values of the SPSR.IT[7:0] bits generated on exception entry are always correct for the preferred exception return, but might require adjustment by software if returning elsewhere.

In some cases, the value of the LR set on taking the exception, as shown in Table B1-4 on page B1-34, makes it necessary to perform a subtraction to calculate the appropriate return address. The value that must be subtracted for the preferred exception return, and other details of the preferred exception return, are given in the description of each of the exceptions.

The ARM architecture provides the following *exception return instructions*:

- Data-processing instructions with the S bit set and the PC as a destination, see *SUBS PC, LR and related instructions* on page B6-25.

  Typically, `SUBS` is used when a subtraction is required, and `SUBS` with an operand of 0 or `MOVS` is used otherwise.

- From ARMv6, the `RFE` instruction, see *RFE* on page B6-16. If a subtraction is required, typically it is performed before saving the LR value to memory.

- In ARM state, a form of the `LDM` instruction, see *LDM (exception return)* on page B6-5. If a subtraction is required, typically it is performed before saving the LR value to memory.

Alignment of exception returns

An unaligned exception return is one where the address transferred to the PC on an exception return is not aligned to the size of instructions in the target instruction set. The target instruction set is controlled by the [J,T] bits of the value transferred to the CPSR for the exception return. The behavior of the hardware for exception returns for different values of the [J,T] bits is as follows:

[J,T] == 00 The target instruction set state is ARM state. Bits [1:0] of the address transferred to the PC are ignored by the hardware.

[J,T] == 01 The target instruction set state is Thumb state:
- bit [0] of the address transferred to the PC is ignored by the hardware
- bit [1] of the address transferred to the PC is part of the instruction address.

[J,T] == 10 The target instruction set state is Jazelle state. In a non-trivial implementation of the Jazelle extension, bits [1:0] of the address transferred to the PC are part of the instruction address. In a trivial implementation of the Jazelle extension, behavior is UNPREDICTABLE, see *Exception return to an unsupported instruction set state* on page B1-40. For details of the trivial implementation of Jazelle state see *Trivial implementation of the Jazelle extension* on page B1-81.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- 例外を引き起こした命令が例外ハンドラでエミュレートされている場合は、次に実行される命令

ＡＲＭアーキテクチャでは、例外からの復帰を実行ストリーム内の特定の場所にしなければならないという要件はありません。ただし、リセット以外の例外には、それぞれ*望ましい例外からの復帰*がアーキテクチャで設定されています。例外の開始時点で生成される SPSR.IT[7:0] ビットの値は、必ず望ましい例外からの復帰に則していますが、他の場所に復帰する場合はソフトウェアによる調整が必要になる場合があります。

P. B1-34 の表 B1-4 に示すように、例外が取得された時点で設定される ＬＲ の値によっては、適切な復帰アドレスを計算するための減算が必要になります。望ましい例外からの復帰のために減算する必要のある値と、その他の望ましい例外からの復帰の詳細については、それぞれの例外の説明の中で示します。

ARM アーキテクチャでは、次の*例外からの復帰命令*が提供されます。

- S ビットがセットされ、PC をデスティネーションとして使用するデータ処理命令。*SUBS PC、LR、および関連する命令* P. B6-25 を参照して下さい。

  通常、減算が必要な場合は `SUBS` が使用され、それ以外の場合はオペランドが 0 の `SUBS` か、`MOVS` が使用されます。

- ARMv6 およびそれ以降の `RFE` 命令。*RFE* P. B6-16 を参照して下さい。通常、減算が必要な場合は、LR の値がメモリに保存される前に減算が実行されます。

- ARM 状態での特定形式の `LDM` 命令。*LDM（例外からの復帰）* P. B6-5 を参照して下さい。通常、減算が必要な場合は、LR の値がメモリに保存される前に減算が実行されます。

例外からの復帰のアライメント

アンアラインな例外からの復帰とは、例外からの復帰時点で PC に転送されるアドレスが、対象の命令セット内の命令サイズにアラインされていない復帰を意味します。対象の命令セットは、例外からの復帰のために CPSR に転送される値の [J,T] ビットによって制御されます。[J,T] ビットの値による、例外からの復帰に関するハードウェアの動作の違いは次のとおりです。

[J,T] = 00 対象の命令セット状態は ARM 状態です。PC に転送されたアドレスのビット [1:0] は、ハードウェアで無視されます。

[J,T] = 01 対象の命令セット状態は Thumb 状態です。
- PC に転送されたアドレスのビット [0] は、ハードウェアで無視されます。
- PC に転送されたアドレスのビット [1] は、命令アドレスの一部になります。

[J,T] = 10 対象の命令セット状態は Jazelle 状態です。Jazelle 拡張機能の非トリビアル実装では、PC に転送されたアドレスのビット [1:0] が命令アドレスの一部になります。Jazelle 拡張機能のトリビアル実装では、動作が予測不能になります。*サポートされていない命令セット状態への例外からの復帰* P. B1-40 を参照して下さい。
Jazelle 状態のトリビアル実装の詳細については、*Jazelle 拡張機能のトリビアル実装* P. B1-81 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

[J,T] == 11 The target instruction set state is ThumbEE state:

- bit [0] of the address transferred to the PC is ignored by the hardware
- bit [1] of the address transferred to the PC is part of the instruction address.

ARM deprecates any dependence on the requirements that the hardware ignores bits of the address. ARM recommends that the address transferred to the PC for an exception return is correctly aligned for the target instruction set.

After an exception entry other than Reset, the LR value has the correct alignment for the instruction set indicated by the SPSR.[J,T] bits. This means that if exception return instructions are used with the LR and SPSR values produced by such an exception entry, the only precaution software needs to take to ensure correct alignment is that any subtraction is of a multiple of four if returning to ARM state, or a multiple of two if returning to Thumb state or to ThumbEE state.

Exception return to an unsupported instruction set state

An implementation that does not support one or both of Jazelle and ThumbEE states does not normally get into an unsupported instruction set state, because:

- on a trivial Jazelle implementation, the `BXJ` instruction acts as a `BX` instruction
- on an implementation that does not include ThumbEE support, the `ENTERX` instruction is UNDEFINED
- normal exception entry and return preserves the instruction set state.

However, it is possible for an exception return instruction to set CPSR.J and CPSR.T to the values corresponding to an unsupported instruction set state. This is most likely to happen because a faulty exception handler restores the wrong value to the CPSR.

If the processor attempts to execute an instruction while the CPSR.J and CPSR.T bits indicate an unsupported instruction set state:

- If the unsupported instruction set state is Jazelle state, behavior is UNPREDICTABLE.
- If the unsupported instruction set state is ThumbEE state, the processor takes an Undefined Instruction exception.

  The Undefined Instruction handler can detect the cause of this exception because on entry to the handler the SPSR.J and SPSR.T bits indicate the ThumbEE state. If the Undefined Instruction handler wants to return, avoiding a return to ThumbEE state, it can change the values its exception return instruction writes to the CPSR.J and CPSR.T bits.

  If an exception return writes CPSR.J = 1 and CPSR.T = 1, corresponding to ThumbEE state, and also writes the address of an aborting memory location to the PC, it is IMPLEMENTATION DEFINED whether:
  - — the instruction is fetched and a Prefetch Abort exception is taken because the memory access aborts
  - — an Undefined Instruction exception is taken, without the instruction being fetched.

An implementation that supports neither of the Jazelle and ThumbEE states can implement the J bits of the PSRs as RAZ/WI. On such an implementation, a return to an unsupported instruction set state cannot occur.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

[J,T] == 11 対象の命令セット状態は ThumbEE 状態です。

- PC に転送されたアドレスのビット [0] は、ハードウェアで無視されます。
- PC に転送されたアドレスのビット [1] は、命令アドレスの一部になります。

ハードウェアでアドレスのビットが無視される要件は一貫している必要があります。例外からの復帰のために PC に転送されるアドレスは、対象の命令セットに正しくアラインすることを推奨します。

リセット以外の例外の開始後に、LR の値は、SPSR.[J,T] ビットで指定された命令セットに適切なアライメントです。つまり、このような例外の開始によって生成された LR と SPSR の値について例外からの復帰命令が使用された場合に、正しいアライメントを保証するためにソフトウェアで実施する必要のある唯一の予防措置は、すべての減算を、ARM 状態に復帰する場合は 4 の倍数にし、Thumb 状態や ThumbEE 状態に復帰する場合は 2 の倍数にすることです。

サポートされていない命令セット状態への例外からの復帰

通常、Jazelle 状態と ThumbEE 状態のどちらかまたは両方をサポートしていない実装では、次のような理由から、サポートされていない命令セット状態には遷移しません。

- トリビアル Jazelle 実装では、`BXJ` 命令が `BX` 命令として機能します。
- ThumbEE をサポートしていない実装では、`ENTERX` 命令が未定義になります。
- 通常の例外の開始と例外からの復帰では、命令セット状態が保存されます。

ただし、例外からの復帰命令によって、CPSR.J と CPSR.T が、サポートされていない命令セット状態に対応する値に設定される場合があります。これはほとんどの場合、欠陥のある例外ハンドラによって、不正な値が CPSR に復元された場合に発生します。

CPSR.J ビットと CPSR.T ビットがサポートされていない命令セット状態を示しているときに、プロセッサが命令を実行しようとした場合、次の動作が行われます。

- サポートされていない命令セット状態が Jazelle 状態の場合、動作は予測不能です。
- サポートされていない命令セット状態が ThumbEE 状態の場合、プロセッサが未定義命令例外を取得します。

  未定義命令ハンドラは、その開始時点で SPSR.J ビットと SPSR.T ビットが ThumbEE 状態を示しているため、この例外の原因を特定することができます。未定義命令ハンドラが ThumbEE 状態以外に復帰する場合は、例外からの復帰命令により CPSR.J ビットと CPSR.T ビットに書き込まれた値を変更することができます。

  例外からの復帰によって ThumbEE 状態に対応する CPSR.J = 1 と CPSR.T = 1 が書き込まれ、アボートするメモリ位置のアドレスが PC に書き込まれた場合、次の動作のどちらが行われるかは実装定義です。
  - — 命令がフェッチされ、メモリアクセスがアボートすることで、プリフェッチアボート例外が取得される。
  - — 命令がフェッチされず、未定義命令例外が取得される。

Jazelle 状態と ThumbEE 状態のいずれもサポートしていない実装では、PSR の J ビットを RAZ/WI として実装することができます。このような実装では、サポートされていない命令セット状態への復帰は発生しません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.6.5 Exception-handling instructions

From ARMv6, the instruction sets include the following exception-handling instructions, in addition to the exception return instructions described in *Exception return* on page B1-38:

- a CPS (Change Processor State) instruction to simplify changes of processor mode and the disabling and enabling of interrupts, see *CPS* on page B6-3
- an SRS (Store Return State) instruction, to reduce the processing cost of handling exceptions in a different mode to the exception entry mode, by removing any need to use the stack of the original mode, see *SRS* on page B6-20.

As an example of where these instructions might be used, an IRQ routine might want to execute in System or Supervisor mode, so that it can both re-enable IRQs and use BL instructions. This is not possible in IRQ mode, because a nested IRQ could corrupt the return link of a BL at any time.

With the CPS and SRS instructions, the system can use the following instruction sequence at the start of its exception handler:

```
SUB    LR,LR,#4        ; IRQ requires subtraction from LR
SRSFD  SP!, #<mode>    ; <mode> = 19 for Supervisor, 31 for System
CPSIE  i,#<mode>
```

This:

- stores the return state held in the LR and SPSR_irq to the stack for Supervisor mode or for User and System mode
- switches to Supervisor or System mode and re-enables IRQs.

This is done efficiently, without making any use of SP_irq or the IRQ stack.

At the end of the exception handler, an RFEFD SP! instruction pops the return state off the stack and returns from the exception.

B1.6.6 Control of exception handling by the Security Extensions

The Security Exceptions provide additional controls of the handling of:

- aborts, see *Control of aborts by the Security Extensions*
- FIQs, see *Control of FIQs by the Security Extensions* on page B1-42
- IRQs, see *Control of IRQs by the Security Extensions* on page B1-43.

Control of aborts by the Security Extensions

The CPSR.A bit can be used to disable asynchronous aborts. When the Security Extensions are implemented:

- the SCR.AW bit controls whether the CPSR.A bit can be modified in Non-secure state
- the SCR.EA bit controls whether external aborts are handled in Abort mode or Monitor mode.

For details of these bits see *c1, Secure Configuration Register (SCR)* on page B3-106.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.6.5 例外処理命令

ARMv6 およびそれ以降では、命令セットに、*例外からの復帰* P. B1-38 で説明されているような例外からの復帰命令に加えて、次の例外処理命令が追加されています。

- プロセッサモードの変更と割り込みの許可 / 禁止を簡素化する CPS（プロセッサ状態変更）命令（*CPS* P. B6-3 を参照）
- 元のモードのスタックを使用する必要性を排除することによって、例外の開始モードとは別のモードで例外の処理コストを削減する SRS（復帰状態ストア）命令（*SRS* P. B6-20 を参照）

これらの命令の使用例として、IRQ ルーチンで IRQ の再許可と BL 命令の使用の両方を実行できるように、IRQ ルーチンをシステムモードまたはスーパーバイザモードで実行する場合が挙げられます。ネストした IRQ の場合は常に BL の復帰リンクが破壊される可能性があるため、IRQ モードでは実行できません。

CPS 命令と SRS 命令を使用すれば、システムは例外ハンドラの先頭で次の命令シーケンスを使用することができます。

```
SUB    LR,LR,#4        ; IRQ requires subtraction from LR
SRSFD  SP!, #<mode>    ; <mode> = 19 for Supervisor, 31 for System
CPSIE  i,#<mode>
```

この命令シーケンスによって次の処理が実行されます。

- スーパーバイザモードまたはユーザモードおよびシステムモード用のスタックに、LR と SPSR_irq に保持されている復帰状態を保存する。
- スーパーバイザモードまたはシステムモードに切り替え、IRQ を再度許可する。

この処理は、SP_irq または IRQ スタックを使用する必要がないため、効率的です。

例外ハンドラの最後で、RFEFD SP! 命令によりスタックから復帰状態をポップして、例外から復帰します。

B1.6.6 セキュリティ拡張機能による例外処理の制御

セキュリティ拡張機能は、次の処理について追加制御を提供します。

- アボート（*セキュリティ拡張機能によるアボートの制御*を参照）
- FIQ（*セキュリティ拡張機能による FIQ の制御* P. B1-42 を参照）
- IRQ（*セキュリティ拡張機能による IRQ の制御* P. B1-43 を参照）

セキュリティ拡張機能によるアボートの制御

CPSR.A ビットを使用して非同期アボートを禁止することができます。セキュリティ拡張機能が実装されている場合は、次の規則が適用されます。

- SCR.AW ビットによって、非セキュア状態で CPSR.A ビットを変更できるかどうかが制御されます。
- SCR.EA ビットによって、外部アボートがアボートモードとモニタモードのどちらで処理されるかが制御されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B1-9 shows the possible values for the SCR.AW and SCR.EA bits, and the abort handling that results in each case:

Table B1-9 Effect of the SCR.AW and SCR.EA bits on abort handling

SCR bits		Effect on abort handling
AW	EA	
0	0	All aborts are handled locally using Abort mode. Asynchronous aborts are maskable only in Secure state. This is the reset state and supports legacy systems.
0	1	All external aborts, synchronous and asynchronous, are handled in Monitor mode. Asynchronous aborts are maskable only in Secure state. All security aborts from peripherals can be treated in a safe manner in Monitor mode.
1	0	All aborts are handled locally, using Abort mode. Asynchronous aborts are maskable in both Secure and Non-secure states.
1	1	All external aborts are trapped to Monitor mode. Non-secure state can hide asynchronous external aborts from the Monitor, by changing the CPSR.A bit.

When the SCR.EA bit is set to 1, and an external abort causes entry to Monitor mode, fault information is written to the Secure copies of the Fault Status and Fault Address registers.

Control of FIQs by the Security Extensions

The CPSR.F bit can be used to disable FIQs. When the Security Extensions are implemented:

- the SCR.FW bit controls whether the CPSR.F bit can be modified in Non-secure state
- the SCR.FIQ bit controls whether FIQs are handled in FIQ mode or Monitor mode.

For details of these bits see *c1, Secure Configuration Register (SCR)* on page B3-106.

Table B1-10 on page B1-43 shows the effect of these bits on FIQ handling:

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

これらのビットの詳細については、*c1*、*セキュア構成レジスタ (SCR)* P. B3-106 を参照して下さい。

SCR.AW ビットと SCR.EA ビットに可能な値と、それぞれで実施されるアボート処理を、表 B1-9 に示します。

表 B1-9 アボート処理に対する SCR.AW ビットと SCR.EA ビットの影響

SCR ビット		アボート処理に対する影響
AW	EA	
0	0	すべてのアボートが、アボートモードを使用してローカルに処理されます。非同期アボートはセキュア状態でのみマスクできます。 これはリセット時の状態で、レガシーシステムに対応したものです。
0	1	同期と非同期を含むすべての外部アボートがモニタモードで処理されます。非同期アボートはセキュア状態でのみマスクできます。 ペリフェラルからのすべてのセキュリティアボートを、モニタモードの安全な方法で処理することができます。
1	0	すべてのアボートが、アボートモードを使用してローカルに処理されます。非同期アボートは、セキュア状態と非セキュア状態の両方でマスクできます。
1	1	すべての外部アボートがモニタモードにトラップされます。CPSR.A ビットを変更することによって、非セキュア状態で非同期外部アボートをモニタから隠すことができます。

SCR.EA ビットが 1 にセットされており、外部アボートによってモニタモードに遷移した場合は、障害情報がフォールトステータス レジスタとフォールトアドレス レジスタのセキュアコピーに書き込まれます。

セキュリティ拡張機能による FIQ の制御

CPSR.F ビットを使用して FIQ を禁止することができます。セキュリティ拡張機能が実装されている場合は、次の規則が適用されます。

- SCR.FW ビットによって、非セキュア状態で CPSR.F ビットを変更できるかどうかが制御されます。
- SCR.FIQ ビットによって、FIQ が FIQ モードとモニタモードのどちらで処理されるかが制御されます。

これらのビットの詳細については、*c1*、*セキュア構成レジスタ (SCR)* P. B3-106 を参照して下さい。

FIQ 処理に対するこれらのビットの影響を、P. B1-43 の表 B1-10 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B1-10 Effect of the SCR.AW and SCR.EA bits on FIQ handling

SCR bits FW	SCR bits FIQ	Effect on FIQ handling
0	0	FIQs are handled locally using FIQ mode. FIQs are maskable only in Secure state. This is the reset state and supports legacy systems.
0	1	FIQs are handled in Monitor mode. FIQs are maskable only in Secure state. This setting gives Secure FIQs.
1	0	FIQs are handled locally in FIQ mode. FIQs can be masked, in both Secure and Non-secure states.
1	1	All FIQs are trapped to Monitor mode. Non-secure state can hide FIQs from the Monitor, by changing the CPSR.F bit.

Note

- The configuration with SCR.FW == 1 and SCR.FIQ == 1 permits Non-secure state to deny service by changing the CPSR.F bit. ARM recommends that this configuration is not used.
- Interrupts driven by Secure peripherals are called Secure interrupts. When SCR.FW = 0 and SCR.FIQ = 1, FIQ exceptions can be used as Secure interrupts. These enter Secure state in a deterministic way.

Control of IRQs by the Security Extensions

When the Security Extensions are implemented, the SCR.IRQ bit controls whether IRQs are handled in IRQ mode or Monitor mode. For details of this bit see *c1, Secure Configuration Register (SCR)* on page B3-106.

B1.6.7 Low interrupt latency configuration

The SCTLR.FI bit is set to 1 to enable the low interrupt latency configuration of an implementation. This configuration can reduce the interrupt latency of the processor. The mechanisms implemented to achieve low interrupt latency are IMPLEMENTATION DEFINED. For the description of the SCTLR see:

- *c1, System Control Register (SCTLR)* on page B3-96 for a VMSA implementation
- *c1, System Control Register (SCTLR)* on page B4-45 for a PMSA implementation.

To ensure that a change between normal and low interrupt latency configurations is synchronized correctly, the SCTLR.FI bit must be changed only in IMPLEMENTATION DEFINED circumstances. The FI bit can be changed shortly after reset and before enabling the MMU, MPU, or caches, when interrupts are disabled, using the following sequence:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B1-10 FIQ 処理に対する SCR.AW ビットと SCR.EA ビットの影響

SCR ビット FW	SCR ビット FIQ	FIQ 処理に対する影響
0	0	FIQ は FIQ モードを使用してローカルに処理されます。FIQ はセキュア状態でのみマスクできます。 これはリセット時の状態で、レガシーシステムに対応したものです。
0	1	FIQ はモニタモードで処理されます。FIQ はセキュア状態でのみマスクできます。 この設定によってセキュア FIQ が提供されます。
1	0	FIQ は FIQ モードでローカルに処理されます。FIQ は、セキュア状態と非セキュア状態の両方でマスクできます。
1	1	すべての FIQ がモニタモードにトラップされます。CPSR.F ビットを変更することによって、非セキュア状態で FIQ をモニタから隠すことができます。

注

- SCR.FW = 1 かつ SCR.FIQ = 1 の構成では、CPSR.F ビットを変更することによって、非セキュア状態でサービスを拒否することができます。この構成は使用しないことをお勧めします。
- セキュアペリフェラルによって駆動された割り込みは、セキュア割り込みと呼ばれます。SCR.FW = 0 かつ SCR.FIQ = 1 の場合、FIQ 例外をセキュア割り込みとして利用することができます。この例外は、決定論的な方法でセキュア状態に遷移します。

セキュリティ拡張機能による IRQ の制御

セキュリティ拡張機能が実装されている場合は、SCR.IRQ ビットによって、IRQ が IRQ モードとモニタモードのどちらで処理されるかが制御されます。これらのビットの詳細については、*c1、セキュア構成レジスタ (SCR)* P. B3-106 を参照して下さい。

B1.6.7 低割り込みレイテンシ構成

実装の低割り込みレイテンシ構成を許可するには、SCTLR.FI ビットを 1 にセットします。この構成では、プロセッサの割り込みレイテンシを短縮することができます。低割り込みレイテンシを実現するために実装される機構は、実装定義です。SCTLR については、次のセクションを参照して下さい。

- *c1、システム制御レジスタ (SCTLR)* P. B3-96（VMSA 実装の場合）
- *c1、システム制御レジスタ (SCTLR)* P. B4-45 （PMSA 実装の場合）

標準割り込みレイテンシ構成と低割り込みレイテンシ構成との間の変更が正しく同期することを保証するには、SCTLR.FI ビットを実装定義の状況でのみ変更する必要があります。FI ビットは、リセット直後から MMU、MPU、またはキャッシュが稼動するまでの、割り込みが不可能な間に、次のシーケンスを使用して変更できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
DSB
ISB
MCR p15, 0, Rx, c1, c0, c0     ; change FI bit in the SCTLR
DSB
ISB
```

Implementation can define other sequences and circumstances that permit the SCTLR.FI bit to be changed.

When interrupt latency is reduced, this can result in reduced performance overall. Examples of methods that might be used to reduce interrupt latency are:

- disabling Hit-Under-Miss functionality in a processor
- the abandoning of restartable external accesses.

These choices permit the processor to react to a pending interrupt faster than would otherwise be the case.

A low interrupt latency configuration permits interrupts and asynchronous aborts to be taken during a sequence of memory transactions generated by a load/store instruction. For details of what these sequences are and the consequences of taking interrupts and asynchronous aborts in this way see *Single-copy atomicity* on page A3-27.

ARM deprecates any software reliance on the behavior that an interrupt or asynchronous abort cannot occur in a sequence of memory transactions generated by a single load/store instruction to Normal memory.

Note

A particular case that has shown this reliance is load multiples that load the stack pointer from memory. In an implementation where an interrupt is taken during the `LDM`, this can result in corruption of the stack pointer.

B1.6.8 Wait For Event and Send Event

A multiprocessor operating system requires locking mechanisms to protect data structures from being accessed simultaneously by multiple processors. These mechanisms prevent the data structures becoming inconsistent or corrupted if different processors try to make conflicting changes. If a lock is busy, because a data structure is being used by one processor, it might not be practical for another processor to do anything except wait for the lock to be released. For example, if a processor is handling an interrupt from a device it might need to add data received from the device to a queue. If another processor is removing data from the queue, it will have locked the memory area that holds the queue. The first processor cannot add the new data until the queue is in a consistent state and the lock has been released. It cannot return from the interrupt handler until the data has been added to the queue, so it must wait.

Typically, a spin-lock mechanism is provided for these circumstances:

- A processor requiring access to the protected data attempts to obtain the lock using single-copy atomic synchronization primitives such as the ARM Load-Exclusive and Store-Exclusive operations described in *Synchronization and semaphores* on page A3-12.
- If the processor obtains the lock it performs its memory operation and releases the lock.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

```
DSB
ISB
MCR p15, 0, Rx, c1, c0, c0     ; change FI bit in the SCTLR
DSB
ISB
```

SCTLR.FI ビットの変更を可能にするその他のシーケンスや状況は、実装で定義することができます。

割り込みレイテンシが短縮されると、全体のパフォーマンスが低下する可能性があります。割り込みレイテンシを短縮するために使用できる手段の例には、次のようなものがあります。

- プロセッサ内のヒット アンダーミス機能を非稼動にする。
- 再始動可能な外部アクセスを破棄する。

これらを選択することによって、プロセッサは、他の場合よりも迅速に保留割り込みに対処することができます。

低割り込みレイテンシ構成では、ロード / ストア命令によって生成されるメモリトランザクション シーケンスの間に、割り込みと非同期アボートを取得することができます。このシーケンスの内容と、このような方法で割り込みと非同期アボートを取得する結果については、*シングルコピーのアトミック性* P. A3-27 を参照して下さい。

ソフトウェアで、ノーマルメモリに対する単一のロード / ストア命令によって生成されるメモリトランザクション シーケンスで割り込みまたは非同期アボートが発生しない動作を前提とすることは推奨されません。

注

この依存関係を示す特殊なケースが、メモリからスタックポインタをロードする複数ロードです。`LDM` 中に割り込みが取得される実装では、これによってスタックポインタが破壊される可能性があります。

B1.6.8 イベント待ちとイベント送信

マルチプロセッサのオペレーティングシステムには、複数のプロセッサによる同時アクセスからデータ構造を保護するロック機構が必要です。この機構によって、複数のプロセッサによる変更が競合した場合のデータ構造の不整合や破損が防止されます。ロックがビジーの場合は、データ構造が 1 つのプロセッサによって使用されているため、他のプロセッサにとって、ロックの解放を待つこと以外の動作は現実的ではありません。例えば、プロセッサがデバイスからの割り込みを処理中に、デバイスから受け取ったデータをキューに追加しなければならない場合があります。他のプロセッサがキューからデータを取り除いている場合、キューが保持されているメモリ領域がロックされています。最初のプロセッサは、キューが整合性の取れた状態になってロックが解放されるまで、新しいデータを追加することができません。また、データがキューに追加されるまで割り込みハンドラから復帰することができないため、待つ以外にありません。

通常、このような状況ではスピンロック機構が提供されます。

- 保護されたデータにアクセスする必要のあるプロセッサは、*同期化とセマフォ* P. A3-12 で説明されているように、ARM の排他ロード動作や排他ストア動作などのシングルコピー アトミックな同期化基本命令を使用してロックの取得を試みます。
- プロセッサがロックを取得すると、メモリ動作を実行してからロックを解放します。
- プロセッサがロックを取得できない場合は、ロックが使用可能になるまで、緊密なループでロックの値の読み出しを繰り返します。ロックが使用可能になった時点で、再度、ロックの取得を試みます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- If the processor cannot obtain the lock, it reads the lock value repeatedly in a tight loop until the lock becomes available. At this point it again attempts to obtain the lock.

However, this spin-lock mechanism is not ideal for all situations:

- in a low-power system the tight read loop is undesirable because it uses energy to no effect
- in a multi-threaded processor the execution of spin-locks by waiting threads can significantly degrade overall performance.

Therefore, ARMv7 provides an alternative locking mechanism based on events. The Wait For Event lock mechanism permits a processor that has failed to obtain a lock to enter a low-power state. When the processor that currently holds the required lock releases the lock it sends an event that causes any waiting processors to wake up and attempt to gain the lock again.

Note

Although a complex operating system can contain thousands of distinct locks, the event sent by this mechanism does not indicate which lock has been released. If the event relates to a different lock, or if another processor acquires the lock more quickly, the processor fails to acquire the lock and can re-enter the low-power state waiting for the next event.

The Wait For Event system relies on hardware and software working together to achieve energy saving:

- the hardware provides the mechanism to enter the Wait For Event low-power state
- the operating system software is responsible for issuing:
  - a Wait For Event instruction when waiting for a spin-lock, to enter the low-power state
  - a Send Event instructions when releasing a spin-lock.

The mechanism depends on the interaction of:

- WFE wake-up events, see *WFE wake-up events*
- the Event Register, see *The Event Register* on page B1-46
- the Send Event instruction, see *The Send Event instruction* on page B1-46
- the Wait For Event instruction, see *The Wait For Event instruction* on page B1-46.

WFE wake-up events

The following events are *WFE wake-up events*:

- the execution of an SEV instruction on any processor in the multiprocessor system
- an IRQ interrupt, unless masked by the CPSR.I bit
- an FIQ interrupt, unless masked by the CPSR.F bit
- an asynchronous abort, unless masked by the CPSR.A bit
- a debug event, if invasive debug is enabled and the debug event is permitted.

For details of the masking bits in the CPSR see *Format of the CPSR and SPSRs* on page B1-16. This masking is an important consideration with this mechanism, because lock mechanisms can be required when interrupts are disabled.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ただし、このスピンロック機構は、すべての状況にとって理想的なわけではありません。

- 低消費電力システムでの緊密な読み出しループは、エネルギーを消費する割に効果がないため、望ましくありません。
- マルチスレッド プロセッサでの待ちスレッドによるスピンロックの実行は、全体のパフォーマンスを大幅に低下させる可能性があります。

このため、ARMv7 では、イベントに基づく別のロック機構が提供されています。イベント待ちロック機構では、ロックの取得に失敗したプロセッサが低電力状態に遷移することができます。現在必要なロックを保持しているプロセッサがロックを解放するときに、待ち状態のプロセッサをウェークアップしてロックの取得を再試行させるイベントを送信します。

注

複雑なオペレーティングシステムには何千もの異なるロックを含めることができますが、この機構によって送信されるイベントでは、どのロックが解放されたかは通知されません。イベントが別のロックに関係している場合、または別のプロセッサの方がより速くロックを取得する場合は、プロセッサが再度ロックの取得に失敗して、次のイベントを待つための低電力状態に遷移する可能性があります。

イベント待ちシステムでは、次のようにハードウェアとソフトウェアが協力してエネルギーの節約を実現することが必要です。

- ハードウェアによって、イベント待ち低電力状態に遷移するための機構が提供されます。
- オペレーティングシステム ソフトウェアは、次の命令を発行します。
  - スピンロックを待っている間に低電力状態に遷移するためのイベント待ち命令
  - スピンロックを解放するときのイベント送信命令

この機構は、次の相互動作に依存します。

- WFE ウェークアップ イベント（*WFE ウェークアップ イベント*を参照）
- イベントレジスタ（*イベントレジスタ* P. B1-46 を参照）
- イベント送信命令（*イベント送信命令* P. B1-46 を参照）
- イベント待ち命令（*イベント待ち命令* P. B1-46）

WFE ウェークアップ イベント

次に示すイベントは、*WFE ウェークアップ イベント*です。

- マルチプロセッサ システム内の任意のプロセッサ上での SEV 命令の実行
- IRQ 割り込み（CPSR.I ビットによってマスクされていない場合）
- FIQ 割り込み（CPSR.F ビットによってマスクされていない場合）
- 非同期アボート（CPSR.A ビットによってマスクされていない場合）
- デバッグイベント（侵襲性デバッグが有効になっており、デバッグイベントが許可されている場合）

CPSR 内のマスクビットの詳細については、*CPSR と SPSR のフォーマット* P. B1-16 を参照して下さい。割り込みが不可能な間にロック機構が要求される可能性があるため、この機構にとってマスク設定は重要な考慮事項です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The Event Register

The Event Register is a single bit register for each processor. When set, an event register indicates that an event has occurred, since the register was last cleared, that might prevent the processor needing to suspend operation on issuing a `WFE` instruction.

The value of the Event Register at reset is UNKNOWN.

The Event Register is set by any WFE wake-up event or by the execution of an exception return instruction. For the definition of exception return instructions see *Exception return* on page B1-38.

The Event Register is cleared only by a Wait For Event instruction.

You cannot read or write the value of the Event Register directly.

The Send Event instruction

The Send Event instruction causes an event to be signaled to all processors in the multiprocessor system. The mechanism used to signal the event to the processors is IMPLEMENTATION DEFINED. The Send Event instruction sets the Event Register.

The Send Event instruction, `SEV`, is available to both unprivileged and privileged code, see *SEV* on page A8-316.

The Wait For Event instruction

The action of the Wait For Event instruction depends on the state of the Event Register:

- If the Event Register is set, the instruction clears the register and returns immediately. Normally, if this happens the processor makes another attempt to claim the lock.
- If the Event Register is clear the processor can suspend execution and enter a low-power state. It can remain in that state until the processor detects a WFE wake-up event or a reset. When the processor detects a WFE wake-up event, or earlier if the implementation chooses, the `WFE` instruction completes.

The Wait For Event instruction, `WFE`, is available to both unprivileged and privileged code, see *WFE* on page A8-808.

The code using the Wait For Event mechanism must be tolerant to spurious wake-up events, including multiple wake ups.

Pseudocode details of the Wait For Event lock mechanism

The `ClearEventRegister()` pseudocode procedure clears the Event Register of the current processor.

The `EventRegistered()` pseudocode function returns TRUE if the Event Register of the current processor is set and FALSE if it is clear:

```
boolean EventRegistered()
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

イベントレジスタ

イベントレジスタは、プロセッサごとの単一ビットレジスタです。イベントレジスタがセットされている場合、最後にレジスタがクリアされてからイベントが発生したことを示しています。これによって、プロセッサは`WFE`命令の発行時に動作を中断する必要がなくなります。

リセット時のイベントレジスタの値は、未知です。

イベントレジスタは、任意の WFE ウェークアップ イベントまたは例外からの復帰命令の実行によってセットされます。例外からの復帰命令の定義については、*例外からの復帰* P. B1-38 を参照して下さい。

イベントレジスタは、イベント待ち命令によってのみクリアされます。

イベントレジスタの値を直接読み書きすることはできません。

イベント送信命令

イベント送信命令は、マルチプロセッサシステム内のすべてのプロセッサに伝達されるイベントを発生させます。プロセッサへのイベントの伝達に使用される機構は、実装定義です。イベント送信命令によってイベントレジスタがセットされます。

イベント送信命令（`SEV`）は、特権コードと非特権コードの両方で使用できます。*SEV* P. A8-316 を参照して下さい。

イベント待ち命令

イベント待ち命令の動作は、イベントレジスタの状態に依存します。

- イベントレジスタがセットされている場合、この命令は、レジスタをクリアして、すぐに復帰します。通常、この場合は、プロセッサがロックの取得を再度試みます。
- イベントレジスタがクリアされている場合、プロセッサは実行を中断して、低電力状態に遷移することができます。プロセッサは、WFE ウェークアップ イベントまたはリセットを検出するまで、その状態に留まることができます。プロセッサが WFE ウェークアップ イベントを検出したとき、または実装で選択されている場合はさらに早く、`WFE`命令が完了します。

イベント待ち命令（`WFE`）は、特権コードと非特権コードの両方で使用できます。*WFE* P. A8-808 を参照して下さい。

イベント待ち機構を使用するコードは、複数のウェークアップを含む偽のウェークアップ イベントに対処できる必要があります。

イベント待ちロック機構の擬似コードの詳細

`ClearEventRegister()` 擬似コードプロシージャは、現在のプロセッサのイベントレジスタをクリアします。

`EventRegistered()` 擬似コード関数は、現在のプロセッサのイベントレジスタがセットされている場合に TRUE、クリアされている場合に FALSE を返します。

```
boolean EventRegistered()
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The `WaitForEvent()` pseudocode procedure optionally suspends execution until a WFE wake-up event or reset occurs, or until some earlier time if the implementation chooses. It is IMPLEMENTATION DEFINED whether restarting execution after the period of suspension causes a `ClearEventRegister()` to occur.

The `SendEvent()` pseudocode procedure sets the Event Register of every processor in the multiprocessor system.

B1.6.9 Wait For Interrupt

Previous versions of the ARM architecture have included a Wait For Interrupt concept, and Wait For Interrupt is a required feature of the architecture from ARMv6. In ARMv7, Wait For Interrupt is supported only through an instruction, `WFI`, that is provided in the ARM and Thumb instruction sets. For more information, see *WFI* on page A8-810.

Note

In ARMv7 the CP15 c7 encoding previously used for WFI is redefined as a NOP, see *CP15 c7, No Operation (NOP)* on page B3-138 and *CP15 c7, Miscellaneous functions* on page B4-72.

When a processor issues a `WFI` instruction it can suspend execution and enter a low-power state. It can remain in that state until the processor detects a reset or one of the following *WFI wake-up events*:

- an IRQ interrupt, regardless of the value of the CPSR.I bit
- an FIQ interrupt, regardless of the value of the CPSR.F bit
- an asynchronous abort, regardless of the value of the CPSR.A bit
- a debug event, when invasive debug is enabled and the debug event is permitted.

When the hardware detects a WFI wake-up event, or earlier if the implementation chooses, the `WFI` instruction completes.

WFI wake-up events cannot be masked by the mask bits in the CPSR.

Note

- Because debug entry is one of the WFI wake-up events, ARM strongly recommends that Wait For Interrupt is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures the intervention of debug while waiting does not significantly change the function of the program being debugged.
- In some previous implementations of Wait For Interrupt, the idle loop is followed by exit functions that must be executed before the interrupt is taken. The operation of Wait For Interrupt remains consistent with this model, and therefore differs from the operation of Wait For Event.
- Some implementations of Wait For Interrupt drain down any pending memory activity before suspending execution. This increases the power saving, by increasing the area over which clocks can be stopped. This operation is not required by the ARM architecture, and code must not rely on Wait For Interrupt operating in this way.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`WaitForEvent()` 擬似コードプロシージャはオプションとして、WFE ウェークアップ イベントまたはリセットが発生するまで、あるいは実装で選択されているより早い時点まで実行を中断します。中断期間後の実行の再開によって `ClearEventRegister()` が呼び出されるかどうかは、実装定義です。

`SendEvent()` 擬似コードプロシージャは、マルチプロセッサシステム内のすべてのプロセッサのイベントレジスタをセットします。

B1.6.9 割り込み待ち

割り込み待ちの概念は以前のバージョンの ARM アーキテクチャから含まれており、ARMv6 およびそれ以降のアーキテクチャでは割り込み待ちが必須機能です。ARMv7 では、割り込み待ちが、ARM 命令セットと Thumb 命令セットで提供される `WFI` 命令を通してのみサポートされます。詳細については、*WFI* P. A8-810 を参照して下さい。

注

ARMv7 では、以前 WFI で使用されていた CP15 c7 エンコードが NOP として再定義されています。*CP15 c7、無操作 (NOP)* P. B3-138 と *CP15 c7、その他の機能* P. B4-72 を参照して下さい。

プロセッサが `WFI` 命令を発行した場合は、実行を中断して低電力状態に遷移することができます。プロセッサは、リセットまたは次の *WFI ウェークアップ イベント*のいずれかを検出するまで、その状態に留まることができます。

- IRQ 割り込み（CPSR.I ビットの値に関係なく）
- FIQ 割り込み（CPSR.F ビットの値に関係なく）
- 非同期アボート（CPSR.A ビットの値に関係なく）
- デバッグイベント（侵襲性デバッグが可能で、デバッグイベントが許可されている場合）

ハードウェアが WFI ウェークアップ イベントを検出したとき、または実装で選択されている場合はより早く、`WFI` 命令が完了します。

WFI ウェークアップ イベントは、CPSR 内のマスクビットでマスクできません。

注

- デバッグの開始は WFI ウェークアップ イベントの 1 つであるため、特定の割り込みイベントの発生を待って次に進むのではなく、割り込み待ちをアイドルループの一部として使用することを強く推奨します。これによって、待機中のデバッグの介入で、デバッグ対象のプログラム機能が大幅に変更されないことが保証されます。
- 一部の以前の実装の割り込み待ちでは、割り込みが取得される前に実行する必要のある終了関数がアイドルループの後に続きます。割り込み待ちの動作は、このモデルとの整合性が維持されている点で、イベント待ちの動作とは異なります。
- 割り込み待ちの実装によっては、実行を中断する前にすべての保留中のメモリ動作を終了させるものがあります。これによって、クロックを停止可能な領域が増えることになり、電力がより節約されます。この動作は ARM アーキテクチャで必須ではないため、コードは割り込み待ちがこのように動作することを前提とできません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Using WFI to indicate an idle state on bus interfaces

A common implementation practice is to complete any entry into power-down routines with a `WFI` instruction. Typically, the `WFI` instruction:

1. forces the suspension of execution, and of all associated bus activity
2. ceases to execute instructions from processor.

The control logic required to do this typically tracks the activity of the bus interfaces of the processor. This means it can signal to an external power controller that there is no ongoing bus activity.

The exact nature of this interface is IMPLEMENTATION DEFINED, but the use of Wait For Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it very suitable as the preferred power-down entry mechanism for future implementations.

Pseudocode details of Wait For Interrupt

The `WaitForInterrupt()` pseudocode procedure optionally suspends execution until a WFI wake-up event or reset occurs, or until some earlier time if the implementation chooses.

B1.6.10 Reset

On an ARM processor, when the Reset input is asserted the processor immediately stops execution of the current instruction. When Reset is de-asserted, the actions described in *Exception entry* on page B1-34 are performed, for the Reset exception. The processor then starts executing code, in Supervisor mode with interrupts disabled. Execution starts from the normal or high reset vector address, `0x00000000` or `0xFFFF0000`, as determined by the reset value of the SCTLR.V bit. This reset value can be determined by an IMPLEMENTATION DEFINED configuration input signal.

Note

- The ARM architecture does not distinguish between multiple levels of reset. A system can provide multiple distinct levels of reset that reset different parts of the system. These all correspond to this single reset exception.
- The reset value of the SCTLR.EE bit can be defined by a configuration input signal. If this is done, that value also applies to the CPSR.E bit on reset. For more information see:
  - *c1, System Control Register (SCTLR)* on page B3-96 for a VMSA implementation
  - *c1, System Control Register (SCTLR)* on page B4-45 for a PMSA implementation.

The following pseudocode describes how this exception is taken:

```
// TakeReset()
// ===========

TakeReset()
    // Enter Supervisor mode and (if relevant) Secure state, and reset CP15. This affects
    // the banked versions and values of various registers accessed later in the code.
    // Also reset other system components.
    CPSR.M = '10011';  // Supervisor mode
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

バスインタフェース上のアイドル状態を示すための WFI の使用

一般的な実装方式の 1 つは、`WFI` 命令を使用して電力オフルーチンへの遷移を完了させることです。通常は、`WFI` 命令で次の処理が実行されます。

1. 実行とすべての関連するバス動作を強制的に中断する。
2. プロセッサからの命令の実行を中断する。

通常は、この処理を実行するために必要な制御ロジックが、プロセッサのバスインタフェースの動作を追跡します。これは、進行中のバス動作が存在しないことを外部の電力コントローラに通知できることを意味します。

このインタフェースの正確な特性は実装定義ですが、完全に実行を中断させる唯一のアーキテクチャ定義の機構としての割り込み待ちの使用は、将来の実装に適した電力オフへの遷移機構として適しています。

割り込み待ちの擬似コードの詳細

`WaitForInterrupt()` 擬似コードプロシージャはオプションとして、WFI ウェークアップ イベントまたはリセットが発生するまで、あるいは実装で選択されている場合はより早く、実行を中断します。

B1.6.10 リセット

ARM プロセッサでは、リセット入力がアサートされた場合は、即座に現在の命令の実行が停止されます。リセットがアサート解除された場合は、リセット例外に対して、*例外の開始* P. B1-34 で説明されている動作が実行されます。その後でプロセッサは、割り込みが禁止されているスーパーバイザモードでコードの実行を開始します。実行は、SCTLR.V ビットのリセット値に従い、`0x00000000` または `0xFFFF0000` のノーマルまたはハイ リセットベクタアドレスから開始されます。このリセット時の値は、実装定義の構成入力信号で決定できます。

注

- ARM アーキテクチャでは、複数のリセットレベルが区別されません。システムは、複数の異なるレベルのリセットを提供し、それぞれシステムの異なる部分をリセットできます。これらのすべてが、この単一のリセット例外に対応します。
- SCTLR.EE ビットのリセット時の値は、構成入力信号で定義することができます。この方法を使用する場合、リセット時にその値が CPSR.E ビットにも適用されます。詳細については、次のセクションを参照して下さい。
  - *c1、システム制御レジスタ (SCTLR)* P. B3-96（VMSA 実装の場合）
  - *c1、システム制御レジスタ (SCTLR)* P. B4-45 （PMSA 実装の場合）

次の擬似コードは、この例外の取得方法を示すものです。

```
// TakeReset()
// ===========

TakeReset()
    // Enter Supervisor mode and (if relevant) Secure state, and reset CP15. This affects
    // the banked versions and values of various registers accessed later in the code.
    // Also reset other system components.
    CPSR.M = '10011';  // Supervisor mode
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
if HaveSecurityExt() then SCR.NS = '0';
ResetCP15Registers();
ResetDebugRegisters();
if HaveAdvSIMDorVFP() then FPEXC.EN = '0';  SUBARCHITECTURE_DEFINED further resetting;
if HaveThumbEE() then TEECR.XED = '0';
if HaveJazelle() then JMCR.JE = '0';  SUBARCHITECTURE_DEFINED further resetting;

// Further CPSR changes: all interrupts disabled, IT state reset, instruction set
// and endianness according to the SCTLR values produced by the above call to
// ResetCP15Registers().
CPSR.I = '1';  CPSR.F = '1';  CPSR.A = '1';
CPSR.IT = '00000000';
CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

// All registers, bits and fields not reset by the above pseudocode or by the
// BranchTo() call below are UNKNOWN bitstrings after reset. In particular, the
// return information registers R14_svc and SPSR_svc have UNKNOWN values, so that
// it is impossible to return from a reset in an architecturally defined way.

// Branch to Reset vector.
BranchTo(ExcVectorBase() + 0);
```

The ARM architecture does not define any way of returning from a reset.

B1.6.11 Undefined Instruction exception

An Undefined Instruction exception might be caused by:

- a coprocessor instruction that is not accessible because of the settings in one or both of:
  - — the Coprocessor Access Control Register, see *c1, Coprocessor Access Control Register (CPACR)* on page B3-104 for a VMSA implementation, or *c1, Coprocessor Access Control Register (CPACR)* on page B4-51 for a PMSA implementation
  - — in an implementation that includes the Security Extensions, the Non-Secure Access Control Register, see *c1, Non-Secure Access Control Register (NSACR)* on page B3-110
- a coprocessor instruction that is not implemented
- an instruction that is UNDEFINED
- an attempt to execute an instruction in an unsupported instruction set state, see *Exception return to an unsupported instruction set state* on page B1-40
- division by zero in an SDIV or UDIV instruction in the ARMv7-R profile when the SCTLR.DZ bit is set to 1, see *c1, System Control Register (SCTLR)* on page B4-45.

The Undefined Instruction exception can be used for:

- software emulation of a coprocessor in a system that does not have the physical coprocessor hardware
- *lazy context switching* of coprocessor registers
- general-purpose instruction set extension by software emulation

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
if HaveSecurityExt() then SCR.NS = '0';
ResetCP15Registers();
ResetDebugRegisters();
if HaveAdvSIMDorVFP() then FPEXC.EN = '0';  SUBARCHITECTURE_DEFINED further resetting;
if HaveThumbEE() then TEECR.XED = '0';
if HaveJazelle() then JMCR.JE = '0';  SUBARCHITECTURE_DEFINED further resetting;

// Further CPSR changes: all interrupts disabled, IT state reset, instruction set
// and endianness according to the SCTLR values produced by the above call to
// ResetCP15Registers().
CPSR.I = '1';  CPSR.F = '1';  CPSR.A = '1';
CPSR.IT = '00000000';
CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1:Thumb
CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

// All registers, bits and fields not reset by the above pseudocode or by the
// BranchTo() call below are UNKNOWN bitstrings after reset. In particular, the
// return information registers R14_svc and SPSR_svc have UNKNOWN values, so that
// it is impossible to return from a reset in an architecturally defined way.

// Branch to Reset vector.
BranchTo(ExcVectorBase() + 0);
```


ARM アーキテクチャでは、リセットからの復帰方法は定義されていません。

B1.6.11 未定義命令例外

未定義命令例外は、次の原因で発生する可能性があります。

- 次のどちらかまたは両方の設定が原因でアクセスできないコプロセッサ命令
  - – コプロセッサアクセス制御レジスタ（VMSA 実装の場合は *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B3-104 を、PMSA 実装の場合は *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B4-51 を参照）
  - – セキュリティ拡張機能と非セキュアアクセス制御レジスタを含む実装（*c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3-110 を参照）
- 実装されていないコプロセッサ命令
- 未定義命令
- サポートされていない命令セット状態での命令実行の試み（*サポートされていない命令セット状態への例外からの復帰* P. B1-40 を参照）
- SCTLR.DZ ビットが 1 に設定されている場合の、ARMv7-R プロファイル内の SDIV 命令または UDIV 命令による 0 除算（*c1、システム制御レジスタ (SCTLR)* P. B4-45 を参照）

未定義命令例外は、次の目的に使用できます。

- 物理的にコプロセッサハードウェアが実装されていないシステムでの、コプロセッサのソフトウェアエミュレーション
- コプロセッサレジスタの*低速コンテキストスイッチ*
- ソフトウェアエミュレーションによる汎用命令セット拡張機能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- signaling an illegal instruction execution
- division by zero errors.

In some coprocessor designs, an internal exceptional condition caused by one coprocessor instruction is signaled asynchronously by refusing to respond to a later coprocessor instruction that belongs to the same coprocessor. In these circumstances, the Undefined Instruction handler must take whatever action is needed to clear the exceptional condition, and then return to the second coprocessor instruction.

Note

The only mechanism to determine the cause of an Undefined Instruction exception is analysis of the instruction indicated by the return link in the LR on exception entry. Therefore it is important that a coprocessor only reports exceptional conditions by generating Undefined Instruction exceptions on its own coprocessor instructions.

The following pseudocode describes how this exception is taken:

```
// TakeUndefInstrException()
// =========================

TakeUndefInstrException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 2 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required return
    // address offsets of 2 or 4 respectively.
    new_lr_value = if CPSR.T == '1' then PC-2 else PC-4;
    new_spsr_value = CPSR;

    // Enter Undefined ('11011') mode, and ensure Secure state if initially in Monitor
    // ('10110') mode. This affects the banked versions of various registers accessed later
    // in the code.
    if CPSR.M == '10110' then SCR.NS = '0';
    CPSR.M = '11011';

    // Write return information to registers, and make further CPSR changes: IRQs disabled,
    // IT state reset, instruction set and endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
    CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

    // Branch to Undefined Instruction vector.
    BranchTo(ExcVectorBase() + 4);
```

The preferred exception return from an Undefined Instruction exception is a return to the instruction that generated the exception. Use the LR and SPSR values generated by the exception entry to produce this return as follows:

- If SPSR.J and SPSR.T are both 0, indicating that the exception occurred in ARM state, use an exception return instruction with a subtraction of 4

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- 不正な命令実行の通知
- 0 による除算のエラー

一部のコプロセッサ設計では、1 つのコプロセッサ命令によって引き起こされた内部の例外状態が、同じコプロセッサに属している後続のコプロセッサ命令に対する応答を拒否することによって、非同期的に通知されます。このような状況では、未定義命令ハンドラが、例外状態を解消するために必要なすべての動作を実行してから、2 番目のコプロセッサ命令に戻る必要があります。

注

未定義命令例外の原因を特定する唯一の機構は、例外の開始時点に LR に存在する復帰リンクによって示される命令の分析です。そのため、コプロセッサが例外的な状況を報告するために未定義命令例外を生成する場合、独自のコプロセッサ命令のみを使用することが重要です。

次の擬似コードは、この例外の取得方法を示すものです。

```
// TakeUndefInstrException()
// =========================

TakeUndefInstrException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 2 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required return
    // address offsets of 2 or 4 respectively.
    new_lr_value = if CPSR.T == '1' then PC-2 else PC-4;
    new_spsr_value = CPSR;

    // Enter Undefined ('11011') mode, and ensure Secure state if initially in Monitor
    // ('10110') mode. This affects the banked versions of various registers accessed later
    // in the code.
    if CPSR.M == '10110' then SCR.NS = '0';
    CPSR.M = '11011';

    // Write return information to registers, and make further CPSR changes: IRQs disabled,
    // IT state reset, instruction set and endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1:Thumb
    CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

    // Branch to Undefined Instruction vector.
    BranchTo(ExcVectorBase() + 4);
```

未定義命令例外からの望ましい例外復帰は、例外を生成した命令に復帰することです。次のように、例外の開始によって生成された LR と SPSR の値を使用してこの復帰を実現します。

- SPSR.J と SPSR.T の両方が 0 で、例外が ARM 状態で発生したことを示している場合は、4 の減算を伴う例外からの復帰命令を使用します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- If SPSR.J and SPSR.T are not both 0, indicating that the exception occurred in Thumb state or ThumbEE state, use an exception return instruction with a subtraction of 2.

For more information, see *Exception return* on page B1-38.

Note

- Undefined Instruction exceptions cannot occur in Jazelle state
- If handling the Undefined Instruction exception requires instruction emulation, followed by return to the next instruction after the instruction that caused the exception, the instruction emulator must use the instruction length to calculate the correct return address, and to calculate the updated values of the IT bits if necessary.

Conditional execution of undefined instructions

The conditional execution rules described in *Conditional execution* on page A8-8 apply to all instructions. This includes UNDEFINED instructions and other instructions that would cause entry to the Undefined Instruction exception.

If such an instruction fails its condition check, the behavior depends on the architecture profile and the potential cause of entry to the Undefined Instruction exception, as follows:

- In the ARMv7-A profile:
  - — If the potential cause is the execution of the instruction itself and depends on data values the instruction reads, the instruction executes as a NOP and does not cause an Undefined Instruction exception.
  - — If the potential cause is the execution of an earlier coprocessor instruction, or the execution of the instruction itself but does not depend on data values the instruction reads, it is IMPLEMENTATION DEFINED whether the instruction executes as a NOP or causes an Undefined Instruction exception.

    An implementation must handle all such cases in the same way.
- In the ARMv7-R profile, the instruction executes as a NOP and does not cause an Undefined Instruction exception.

Note

Before ARMv7, all implementations executed any instruction that failed its condition check as a NOP, even if it would otherwise have caused an Undefined Instruction exception. Undefined Instruction handlers written for these implementations might assume without checking that the undefined instruction passed its condition check. Such Undefined Instruction handlers are likely to need rewriting, to check the condition is passed, before they function correctly on all ARMv7-A implementations.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- SPSR.J と SPSR.T の両方が 0 ではなく、例外が Thumb 状態または ThumbEE 状態で発生したことを示している場合は、2 の減算を伴う例外からの復帰命令を使用します。

詳細については、*例外からの復帰* P. B1-38 を参照して下さい。

注

- 未定義命令例外は Jazelle 状態では発生しません。
- 未定義命令例外の処理で、命令エミュレーションを実行してから、例外を引き起こした命令の次の命令に復帰する必要がある場合は、命令エミュレータが命令長を使用して正しい復帰アドレスを計算し、必要に応じて、IT ビットの更新された値を計算する必要があります。

未定義命令の条件付き実行

条件付き実行 P. A8-8 で説明されている条件付き実行の規則が、すべての命令に適用されます。これには、未定義命令と、未定義命令例外への遷移を引き起こす可能性のあるその他の命令が含まれます。

このような命令が条件チェックに失敗した場合の動作は、次のように、アーキテクチャのプロファイルと、未定義命令例外に遷移する潜在的原因によって異なります。

- ARMv7-A プロファイルの場合、次の規則が適用されます。
  - – 潜在的原因が命令の実行そのものであり、命令によって読み出されたデータの値に依存する場合は、命令が NOP として実行され、未定義命令例外は引き起こされません。
  - – 潜在的原因が、以前のコプロセッサ命令の実行、または、命令の実行そのものであるが、命令によって読み出されたデータの値に依存しない場合は、命令が NOP として実行されるか、未定義命令例外が引き起こされるかは実装定義です。

    実装では、このようなすべてのケースを同じ方法で処理する必要があります。
- ARMv7-R プロファイルでは、命令が NOP として実行され、未定義命令例外は引き起こされません。

注

ARMv7 以前は、すべての実装で、未定義命令例外を引き起こしていた可能性があっても、条件チェックに失敗したすべての命令が NOP として実行されていました。このような実装用に作成された未定義命令ハンドラは、未定義命令が条件チェックをパスしたかどうかのチェックを行わず、パスしたことを前提にしている可能性があります。このような未定義命令ハンドラは、条件が満たされているかどうかをチェックするように作り直さなければ、すべての ARMv7-A 実装で正しく機能しない可能性があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.6.12 Supervisor Call (SVC) exception

The Supervisor Call instruction SVC enters Supervisor mode and requests a supervisor function. Typically, the SVC instruction is used to request an operating system function. For more information, see *SVC (previously SWI)* on page A8-430.

Note

In previous versions of the ARM architecture, the SVC instruction was called SWI, Software Interrupt.

The following pseudocode describes how this exception is taken:

```
// TakeSVCException()
// ==================

TakeSVCException()
    // Determine return information. SPSR is to be the current CPSR, after changing the IT[]
    // bits to give them the correct values for the following instruction, and LR is to be
    // the current PC minus 2 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address of
    // the next instruction (the SVC instruction having size 2 or 4 bytes respectively).
    ITAdvance();
    new_lr_value = if CPSR.T == '1' then PC-2 else PC-4;
    new_spsr_value = CPSR;

    // Enter Supervisor ('10011') mode, and ensure Secure state if initially in Monitor
    // ('10110') mode. This affects the banked versions of various registers accessed later
    // in the code.
    if CPSR.M == '10110' then SCR.NS = '0';
    CPSR.M = '10011';

    // Write return information to registers, and make further CPSR changes: IRQs disabled,
    // IT state reset, instruction set and endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
    CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

    // Branch to SVC vector.
    BranchTo(ExcVectorBase() + 8);
```

The preferred exception return from an SVC exception is a return to the next instruction after the SVC instruction. Use the LR and SPSR values generated by the exception entry to produce this return by using an exception return instruction without a subtraction.

For more information, see *Exception return* on page B1-38.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B1.6.12 スーパバイザコール (SVC) 例外

スーパーバイザコール命令（SVC）は、スーパーバイザモードに遷移して、スーパーバイザ機能を要求します。通常、SVC 命令は、オペレーティングシステム機能を要求するために使用されます。詳細については、*SVC（以前の SWI）* P. A8-430 を参照して下さい。

注

以前のバージョンの ARM アーキテクチャでは、SVC 命令が SWI（ソフトウェア割り込み）と呼ばれていました。

次の擬似コードは、この例外の取得方法を示すものです。

```
// TakeSVCException()
// ==================

TakeSVCException()
    // Determine return information. SPSR is to be the current CPSR, after changing the IT[]
    // bits to give them the correct values for the following instruction, and LR is to be
    // the current PC minus 2 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address of
    // the next instruction (the SVC instruction having size 2 or 4 bytes respectively).
    ITAdvance();
    new_lr_value = if CPSR.T == '1' then PC-2 else PC-4;
    new_spsr_value = CPSR;

    // Enter Supervisor ('10011') mode, and ensure Secure state if initially in Monitor
    // ('10110') mode. This affects the banked versions of various registers accessed later
    // in the code.
    if CPSR.M == '10110' then SCR.NS = '0';
    CPSR.M = '10011';

    // Write return information to registers, and make further CPSR changes: IRQs disabled,
    // IT state reset, instruction set and endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1:Thumb
    CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

    // Branch to SVC vector.
    BranchTo(ExcVectorBase() + 8);
```

SVC 例外からの望ましい例外復帰は、SVC 命令の次の命令に復帰することです。例外の開始によって生成された LR と SPSR の値と、減算を伴わない例外からの復帰命令を使用して、この復帰を実現します。

詳細については、*例外からの復帰* P. B1-38 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.6.13 Secure Monitor Call (SMC) exception

The Secure Monitor Call instruction SMC enters Monitor mode and requests a Monitor function. For more information, see *SMC (previously SMI)* on page B6-18.

Note

In previous versions of the ARM architecture, the SMC instruction was called SMI, Software Monitor Interrupt.

The following pseudocode describes how this exception is taken:

```
// TakeSMCException()
// ==================

TakeSMCException()
    // Determine return information. SPSR is to be the current CPSR, after changing the IT[]
    // bits to give them the correct values for the following instruction, and LR is to be
    // the current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address of
    // the next instruction (with the SMC instruction always being 4 bytes in length).
    ITAdvance();
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;

    // Enter Monitor ('10110') mode, and ensure Secure state if initially in Monitor mode.
    // This affects the banked versions of various registers accessed later in the code.
    if CPSR.M == '10110' then SCR.NS = '0';
    CPSR.M = '10110';

    // Write return information to registers, and make further CPSR changes: interrupts
    // disabled, IT state reset, instruction set and endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';  CPSR.F = '1';  CPSR.A = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
    CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

    // Branch to SMC vector.
    BranchTo(MVBAR + 8);
```

The preferred exception return from an SMC exception is a return to the next instruction after the SMC instruction. Use the LR and SPSR values generated by the exception entry to produce this return by using an exception return instruction without a subtraction.

For more information, see *Exception return* on page B1-38.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.6.13 セキュア モニタコール (SMC) 例外

セキュア モニタコール命令 (SMC) は、モニタモードに遷移して、モニタ機能を要求します。詳細については、*SMC（以前の SMI）* P. B6-18 を参照して下さい。

注

以前のバージョンの ARM アーキテクチャでは、SMC 命令が SMI（ソフトウェアモニタ割り込み）と呼ばれていました。

次の擬似コードは、この例外の取得方法を示すものです。

```
// TakeSMCException()
// ==================

TakeSMCException()
    // Determine return information. SPSR is to be the current CPSR, after changing the IT[]
    // bits to give them the correct values for the following instruction, and LR is to be
    // the current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address of
    // the next instruction (with the SMC instruction always being 4 bytes in length).
    ITAdvance();
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;

    // Enter Monitor ('10110') mode, and ensure Secure state if initially in Monitor mode.
    // This affects the banked versions of various registers accessed later in the code.
    if CPSR.M == '10110' then SCR.NS = '0';
    CPSR.M = '10110';

    // Write return information to registers, and make further CPSR changes: interrupts
    // disabled, IT state reset, instruction set and endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';  CPSR.F = '1';  CPSR.A = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1:Thumb
    CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

    // Branch to SMC vector.
    BranchTo(MVBAR + 8);
```

SMC 例外からの望ましい例外復帰は、SMC 命令の次の命令に復帰することです。例外の開始によって生成された LR と SPSR の値と、減算を伴わない例外からの復帰命令を使用して、この復帰を実現します。

詳細については、*例外からの復帰* P. B1-38 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

You can return to the SMC instruction itself by returning using a subtraction of 4, without any adjustment to the SPSR.IT[7:0] bits. The result is that the return occurs, then interrupts or external aborts might occur and be handled, then the SMC instruction is re-executed and another SMC exception occurs.

This relies on:

- the SMC instruction being used correctly, either outside an IT block or as the last instruction in an IT block, so that the SPSR.IT[7:0] bits indicate unconditional execution
- the SMC handler not changing the result of the original conditional execution test for the SMC instruction.

B1.6.14 Prefetch Abort exception

A Prefetch Abort exception can be generated by:

- A synchronous memory abort on an instruction fetch.

  Note

  Asynchronous aborts on instruction fetches are reported using the Data Abort exception, see *Data Abort exception* on page B1-55.

  Prefetch Abort exception entry is synchronous to the instruction whose instruction fetch aborted. If an implementation prefetches instructions, it must handle a synchronous abort on an instruction prefetch by:

  — generating a Prefetch Abort exception if and when the instruction is about to execute

  — ignoring the abort if the instruction does not reach the point of being about to execute, for example, if a branch misprediction or exception entry occurs before the instruction is reached.

  For more information about memory aborts see:

  — *VMSA memory aborts* on page B3-40

  — *PMSA memory aborts* on page B4-13.

- A Breakpoint, Vector Catch or BKPT Instruction debug event, see *Debug exception on Breakpoint, BKPT Instruction or Vector Catch debug events* on page C4-2.

The following pseudocode describes how this exception is taken:

```
// TakePrefetchAbortException()
// ============================

TakePrefetchAbortException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the current instruction plus 4.
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

SPSR.IT[7:0] ビットを調整せずに、4 の減算を使用して復帰することによって、SMC 命令そのものに復帰することができます。この結果、復帰が実行され、割り込みまたは外部アボートが発生した場合は処理され、その後で SMC 命令が再実行されて、別の SMC 例外が発生します。

この動作は次の条件が前提となります。

- SMC 命令が IT ブロックの外部で、または IT ブロック内の最後の命令として正しく使用されており、SPSR.IT[7:0] ビットが無条件実行を示している。
- SMC ハンドラが、SMC 命令の元の条件付き実行テストの結果を変更していない。

B1.6.14 プリフェッチアボート例外

プリフェッチアボート例外は、次の原因で生成される可能性があります。

- 命令フェッチでの同期メモリアボート

  注

  命令フェッチでの非同期アボートは、データアボート例外を使用して報告されます。*データアボート例外* P. B1-55 を参照して下さい。

  プリフェッチアボート例外の開始は、命令フェッチがアボートされた命令と同期します。実装で命令をプリフェッチする場合は、次の方法で命令フェッチでの同期アボートを処理する必要があります。

  – 命令が実行される場合、その直前にプリフェッチアボート例外を生成する。

  – 命令が実行直前のポイントに到達していない場合、例えばその命令に到達する前に分岐予測ミスまたは例外の開始が発生した場合、アボートを無視する。

  メモリアボートの詳細については、次のセクションを参照して下さい。

  – *VMSA メモリアボート* P. B3-40

  – *PMSA メモリアボート* P. B4-13

- ブレークポイント、ベクタキャッチ、または BKPT 命令デバッグイベント。*ブレークポイント、BKPT 命令、ベクタキャッチデバッグ イベントでのデバッグ例外* P. C4-2 を参照して下さい。

次の擬似コードは、この例外の取得方法を示すものです。


```
// TakePrefetchAbortException()
// ============================

TakePrefetchAbortException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the current instruction plus 4.
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
new_spsr_value = CPSR;

// Determine whether this is an external abort to be trapped to Monitor mode.
trap_to_monitor = HaveSecurityExt() && SCR.EA == '1' && IsExternalAbort();

// Enter Abort ('10111') or Monitor ('10110') mode, and ensure Secure state if
// initially in Monitor mode. This affects the banked versions of various registers
// accessed later in the code.
if CPSR.M == '10110' then SCR.NS = '0';
CPSR.M = if trap_to_monitor then '10110' else '10111';

// Write return information to registers, and make further CPSR changes: IRQs disabled,
// other interrupts disabled if appropriate, IT state reset, instruction set and
// endianness to SCTLR-configured values.
SPSR[] = new_spsr_value;
R[14] = new_lr_value;
CPSR.I = '1';
if trap_to_monitor then
    CPSR.F = '1';  CPSR.A = '1';
else
    if !HaveSecurityExt() || SCR.NS == '0' || SCR.AW == '1' then CPSR.A = '1';
CPSR.IT = '00000000';
CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

// Branch to correct Prefetch Abort vector.
if trap_to_monitor then
    BranchTo(MVBAR + 12);
else
    BranchTo(ExcVectorBase() + 12);
```

The preferred exception return from a Prefetch Abort exception is a return to the aborted instruction. Use the LR and SPSR values generated by the exception entry to produce this return by using an exception return instruction with a subtraction of 4.

For more information, see *Exception return* on page B1-38.

B1.6.15 Data Abort exception

A Data Abort exception can be generated by:

- A synchronous abort on a data read or write memory access. Exception entry is synchronous to the instruction that generated the memory access.
- An asynchronous abort. The memory access that caused the abort can be any of:
  - — a data read or write access
  - — an instruction fetch or prefetch
  - — in a VMSA memory system, a translation table access.

  Exception entry occurs asynchronously. It is similar to an interrupt, but uses either Abort mode or Monitor mode, and the associated banked registers. Setting the CPSR.A bit prevents asynchronous aborts from occurring.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
new_spsr_value = CPSR;

// Determine whether this is an external abort to be trapped to Monitor mode.
trap_to_monitor = HaveSecurityExt() && SCR.EA == '1' && IsExternalAbort();

// Enter Abort ('10111') or Monitor ('10110') mode, and ensure Secure state if
// initially in Monitor mode. This affects the banked versions of various registers
// accessed later in the code.
if CPSR.M == '10110' then SCR.NS = '0';
CPSR.M = if trap_to_monitor then '10110' else '10111';

// Write return information to registers, and make further CPSR changes: IRQs disabled,
// other interrupts disabled if appropriate, IT state reset, instruction set and
// endianness to SCTLR-configured values.
SPSR[] = new_spsr_value;
R[14] = new_lr_value;
CPSR.I = '1';
if trap_to_monitor then
    CPSR.F = '1';  CPSR.A = '1';
else
    if !HaveSecurityExt() || SCR.NS == '0' || SCR.AW == '1' then CPSR.A = '1';
CPSR.IT = '00000000';
CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1:Thumb
CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

// Branch to correct Prefetch Abort vector.
if trap_to_monitor then
    BranchTo(MVBAR + 12);
else
    BranchTo(ExcVectorBase() + 12);
```


プリフェッチアボート例外からの望ましい例外復帰は、アボートされた命令に復帰することです。例外の開始によって生成された LR と SPSR の値と、4 の減算を伴う例外からの復帰命令を使用して、この復帰を実現します。

詳細については、*例外からの復帰* P. B1-38 を参照して下さい。

B1.6.15 データアボート例外

データアボート例外は、次の原因で生成される可能性があります。

- データ読み出しまたは書き込みメモリアクセスでの同期アボート。例外の開始は、メモリアクセスを生成した命令と同期します。
- 非同期アボート。このアボートを引き起こす可能性のあるメモリアクセスは次のとおりです。
  - – データ読み出しまたは書き込みアクセス
  - – 命令のフェッチまたはプリフェッチ
  - – VMSA メモリシステムでの変換テーブルアクセス

  例外の開始は非同期に発生します。割り込みに似ていますが、アボートモードとモニタモードのいずれかと、関連するバンクレジスタが使用されます。CPSR.A ビットをセットすると、非同期アボートは発生しなくなります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

There are no asynchronous internal aborts in ARMv7 and earlier architecture versions, so asynchronous aborts are always asynchronous external aborts.

- A Watchpoint debug event, see *Debug exception on Watchpoint debug event* on page C4-3.

Note

A Data Abort exception generated by a Watchpoint debug event can be either asynchronous or synchronous, but is not an abort. This means that if it is asynchronous, it is an asynchronous Data Abort exception but not an asynchronous abort.

For more information about memory aborts see:

- *VMSA memory aborts* on page B3-40
- *PMSA memory aborts* on page B4-13.

The following pseudocode describes how this exception is taken:

```
// TakeDataAbortException()
// ========================

TakeDataAbortException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC plus 4 for Thumb or 0 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the current instruction plus 8. For an asynchronous abort, the PC and CPSR are
    // considered to have already moved on to their values for the instruction following
    // the instruction boundary at which the exception occurred.
    new_lr_value = if CPSR.T == '1' then PC+4 else PC;
    new_spsr_value = CPSR;

    // Determine whether this is an external abort to be trapped to Monitor mode.
    trap_to_monitor = HaveSecurityExt() && SCR.EA == '1' && IsExternalAbort();

    // Enter Abort ('10111') or Monitor ('10110') mode, and ensure Secure state if
    // initially in Monitor mode. This affects the banked versions of various registers
    // accessed later in the code.
    if CPSR.M == '10110' then SCR.NS = '0';
    CPSR.M = if trap_to_monitor then '10110' else '10111';

    // Write return information to registers, and make further CPSR changes: IRQs disabled,
    // other interrupts disabled if appropriate, IT state reset, instruction set and
    // endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';
    if trap_to_monitor then
        CPSR.F = '1';  CPSR.A = '1';
    else
        if !HaveSecurityExt() || SCR.NS == '0' || SCR.AW == '1' then CPSR.A = '1';
    CPSR.IT = '00000000';
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

ARMv7 およびそれ以前のアーキテクチャバージョンでは、非同期内部アボートが存在しないため、非同期アボートは必ず非同期外部アボートです。

- ウォッチポイント デバッグイベント(*ウォッチポイントデバッグ イベントでのデバッグ例外* P. C4-3 を参照)

注

ウォッチポイント デバッグイベントによって生成されたデータアボート例外は、非同期または同期になる可能性がありますが、アボートではありません。これは、非同期の場合は、非同期アボートではなく、非同期データアボート例外になることを意味します。

メモリアボートの詳細については、次のセクションを参照して下さい。

- *VMSA メモリアボート* P. B3-40
- *PMSA メモリアボート* P. B4-13

次の擬似コードは、この例外の取得方法を示すものです。

```
// TakeDataAbortException()
// ========================

TakeDataAbortException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC plus 4 for Thumb or 0 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the current instruction plus 8. For an asynchronous abort, the PC and CPSR are
    // considered to have already moved on to their values for the instruction following
    // the instruction boundary at which the exception occurred.
    new_lr_value = if CPSR.T == '1' then PC+4 else PC;
    new_spsr_value = CPSR;

    // Determine whether this is an external abort to be trapped to Monitor mode.
    trap_to_monitor = HaveSecurityExt() && SCR.EA == '1' && IsExternalAbort();

    // Enter Abort ('10111') or Monitor ('10110') mode, and ensure Secure state if
    // initially in Monitor mode. This affects the banked versions of various registers
    // accessed later in the code.
    if CPSR.M == '10110' then SCR.NS = '0';
    CPSR.M = if trap_to_monitor then '10110' else '10111';

    // Write return information to registers, and make further CPSR changes: IRQs disabled,
    // other interrupts disabled if appropriate, IT state reset, instruction set and
    // endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';
    if trap_to_monitor then
        CPSR.F = '1';  CPSR.A = '1';
    else
        if !HaveSecurityExt() || SCR.NS == '0' || SCR.AW == '1' then CPSR.A = '1';
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

// Branch to correct Data Abort vector.
if trap_to_monitor then
    BranchTo(MVBAR + 16);
else
    BranchTo(ExcVectorBase() + 16);
```

The preferred exception return from a Data Abort exception is a return to the instruction that generated the aborting memory access, or to the instruction following the instruction boundary at which an asynchronous Data Abort exception occurred. Use the LR and SPSR values generated by the exception entry to produce this return by using an exception return instruction with a subtraction of 8.

For more information, see *Exception return* on page B1-38.

Effects of data-aborted instructions

Instructions that access data memory can modify memory by storing one or more values. If a Data Abort exception is generated by executing such an instruction, the value of each memory location that the instruction stores to is:

- unchanged if the memory system does not permit write access to the memory location
- UNKNOWN otherwise.

Instructions that access data memory can modify registers in the following ways:

- By loading values into one or more of the general-purpose registers. The registers loaded can include the PC.
- By specifying *base register write-back*, in which the base register used in the address calculation has a modified value written to it. All instructions that support base register write-back have UNPREDICTABLE results if this is specified with the PC as the base register. Only general-purpose registers other than the PC can be modified reliably in this way.
- By loading values into coprocessor registers.
- By modifying the CPSR.

If a synchronous Data Abort exception is generated by executing such an instruction, the following rules determine the values left in these registers:

1. On entry to the Data Abort handler:
   - the PC value is the Data Abort vector address, see *Exception vectors and the exception base address* on page B1-30
   - the LR_abt value is determined from the address of the aborted instruction.

   Neither value is affected in any way by the results of any load specified by the instruction.
2. The base register is restored to its original value if either:
   - the aborted instruction is a load that includes the base register in the list to be loaded

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
CPSR.IT = '00000000';
CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1:Thumb
CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

// Branch to correct Data Abort vector.
if trap_to_monitor then
    BranchTo(MVBAR + 16);
else
    BranchTo(ExcVectorBase() + 16);
```

データアボート例外からの望ましい例外復帰は、アボートしたメモリアクセスを生成した命令、または非同期データアボート例外が発生した命令境界に続く命令に復帰することです。例外の開始によって生成された LR と SPSR の値と、8 の減算を伴う例外からの復帰命令を使用して、この復帰を実現します。

詳細については、*例外からの復帰* P. B1-38 を参照して下さい。

データアボートされた命令の影響

データメモリにアクセスする命令は、1 つ以上の値をストアしてメモリを変更することができます。このような命令の実行によってデータアボート例外が生成された場合は、命令によってストアされる各メモリ位置の値が次のように処理されます。

- メモリシステムがメモリ位置への書き込みを許可していない場合は変更されません。
- それ以外の場合は未知になります。

データメモリにアクセスする命令は、次の方法でレジスタを変更することができます。

- 1 つ以上の汎用レジスタに値をロードする。ロードするレジスタには、PC を含めることができます。
- アドレス計算で使用されるベースレジスタに変更された値が書き込まれる*ベースレジスタ ライトバック*を指定する。PC をベースレジスタとして使用して、この指定を行った場合は、ベースレジスタ ライトバックをサポートするすべての命令の結果が予測不能になります。この方法で確実に変更できるのは、PC 以外の汎用レジスタのみです。
- コプロセッサレジスタに値をロードする。
- CPSR を変更する。

このような命令を実行することによって、同期データアボート例外が生成された場合は、次の規則によってこれらのレジスタに残される値が決定されます。

1. データアボート ハンドラの開始時には、次の規則が適用されます。
   - PC の値はデータアボート ベクタアドレスです。*例外ベクタと例外ベースアドレス* P. B1-30 を参照して下さい。
   - LR_abt の値は、アボートされた命令のアドレスから決定されます。

   どちらの値も、命令で指定されたロードの結果の影響を受けません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- the base register is being written back.

3. If the instruction only loads one general-purpose register, the value in that register is unchanged.
4. If the instruction loads more than one general-purpose register, UNKNOWN values are left in destination registers other than the PC and the base register of the instruction.
5. If the instruction loads coprocessor registers, UNKNOWN values are left in the destination coprocessor registers.
6. CPSR bits that are not defined as updated on exception entry retain their current value.
7. If a synchronous Data Abort exception is generated by execution of a STREX, STREXB, STREXH, or STREXD instruction:
   - memory is not updated
   - <Rd> is not updated
   - it is UNPREDICTABLE whether the monitor changes from the Exclusive state to the Open state.

The ARM abort model

The abort model used by an ARM processor implementation is described as a *Base Restored Abort Model*. This means that if a synchronous Data Abort exception is generated by executing an instruction that specifies base register write-back, the value in the base register is unchanged.

Note

In versions of the ARM architecture before ARMv6, it is IMPLEMENTATION DEFINED whether the abort model used is the *Base Restored Abort Model* or the *Base Updated Abort Model*. For more information, see *The ARM abort model* on page AppxH-20.

The abort model applies uniformly across all instructions.

B1.6.16 IRQ exception

The IRQ exception is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an IRQ interrupt request input to the processor.

Whether and how an IRQ exception is taken depends on the CPSR.I and SCTLR.FI bits:

- If CPSR.I == 1, IRQ exceptions are disabled and are not taken.
- If CPSR.I == 0 and SCTLR.FI == 0, IRQ exceptions can be taken. In this case IRQ exception entry is precise to an instruction boundary.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

2. 次のいずれかの場合は、ベースレジスタが元の値に復元されます。
   - アボートされた命令がロード命令で、ロード対象のリストにベースレジスタが含まれている。
   - ベースレジスタのライトバックが指定されている。
3. 命令で 1 つの汎用レジスタのみがロードされる場合は、そのレジスタの値は変更されません。
4. 命令で複数の汎用レジスタがロードされる場合は、PC とその命令のベースレジスタ以外のデスティネーションレジスタに未知の値が残されます。
5. 命令でコプロセッサレジスタがロードされる場合は、デスティネーションのコプロセッサレジスタに未知の値が残されます。
6. 例外の開始で更新されるように定義されていない CPSR ビットには、現在の値が保持されます。
7. STREX、STREXB、STREXH、STREXD の各命令の実行によって同期データアボート例外が生成された場合は、次の規則が適用されます。
   - メモリは更新されません。
   - <Rd> は更新されません。
   - モニタが排他的状態からオープンな状態に変化するかどうかは予測不能です。

ARM アボートモデル

ARM プロセッサ実装で使用されるアボートモデルは、*ベース復元アボートモデル*と呼ばれます。これは、ベースレジスタ ライトバックを指定する命令の実行によって、同期データアボート例外が生成された場合は、そのベースレジスタ内の値が変更されないことを意味します。

注

ARMv6 以前のバージョンの ARM アーキテクチャでは、使用されるアボートモデルが*ベース復元アボートモデル*と*ベース更新アボートモデル*のどちらであるかは実装定義です。詳細については、*ARM アボートモデル* P. 付録 H-20 を参照して下さい。

アボートモデルは、すべての命令に同一に適用されます。

B1.6.16 IRQ 例外

IRQ 例外は、実装定義の手段によって生成されます。通常は、プロセッサへの IRQ 割り込み要求入力をアサートすることによって生成されます。

IRQ 例外を取得するか、どのように取得するかは、CPSR.I ビットと SCTLR.FI ビットに依存します。

- CPSR.I == 1 の場合は、IRQ 例外が禁止され、取得されません。
- CPSR.I == 0 かつ SCTLR.FI == 0 の場合は、IRQ 例外を取得できます。この場合は、IRQ 例外の開始は正確に命令境界です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- If CPSR.I == 0 and SCTLR.FI == 1, IRQ exceptions can be taken. In this case IRQ exception entry is precise to an instruction boundary, except that some of the effects of the instruction that follows that boundary might have occurred. These effects are restricted to those that can be repeated idempotently and without breaking the rules in *Single-copy atomicity* on page A3-27. Examples of such effects are:
  - — changing the value of a register that the instruction writes but does not read
  - — performing an access to Normal memory.

  Note

  This relaxation of the normal definition of a precise asynchronous exception permits interrupts to occur during the execution of instructions that change register or memory values, while only requiring the implementation to restore those register values that are needed to correctly re-execute the instruction after the preferred exception return. LDM and STM are examples of such instructions.

The following pseudocode describes how this exception is taken:

```
// TakeIRQException()
// ==================

TakeIRQException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the instruction boundary at which the interrupt occurred plus 4. For this
    // purpose, the PC and CPSR are considered to have already moved on to their values
    // for the instruction following that boundary.
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;

    // Determine whether IRQs are trapped to Monitor mode.
    trap_to_monitor = HaveSecurityExt() && SCR.IRQ == '1';

    // Enter IRQ ('10010') or Monitor ('10110') mode, and ensure Secure state if initially
    // in Monitor mode. This affects the banked versions of various registers accessed
    // later in the code.
    if CPSR.M == '10110' then SCR.NS = '0';
    CPSR.M = if trap_to_monitor then '10110' else '10010';

    // Write return information to registers, and make further CPSR changes: IRQs disabled,
    // other interrupts disabled if appropriate, IT state reset, instruction set and
    // endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';
    if trap_to_monitor then
        CPSR.F = '1';  CPSR.A = '1';
    else
        if !HaveSecurityExt() || SCR.NS == '0' || SCR.AW == '1' then CPSR.A = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
    CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- CPSR.I == 0 かつ SCTLR.FI == 1 の場合は、IRQ 例外を取得できます。この場合は、IRQ 例外の開始は正確に命令境界ですが、その境界に従う命令の影響のうち一部が発生した可能性があります。この影響は、べキ等に繰り返すことが可能で、*シングルコピーのアトミック性* P. A3-27 内の規則に違反しないものに限定されます。このような影響の例には、次のようなものがあります。
  - — 命令によって書き込まれるが読み出されないレジスタの値の変更
  - — ノーマルメモリへのアクセスの実行

  注

  正確な非同期例外の通常の定義をこのように緩和することで、レジスタまたはメモリの値を変更する命令の実行中に割り込みの発生が許可されます。実装で復元する必要があるのは、望ましい例外からの復帰の後に命令を正しく再実行するために必要なレジスタの値のみです。このような命令の例には、LDM および STM があります。

次の擬似コードは、この例外の取得方法を示すものです。

```
// TakeIRQException()
// ==================

TakeIRQException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the instruction boundary at which the interrupt occurred plus 4. For this
    // purpose, the PC and CPSR are considered to have already moved on to their values
    // for the instruction following that boundary.
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;

    // Determine whether IRQs are trapped to Monitor mode.
    trap_to_monitor = HaveSecurityExt() && SCR.IRQ == '1';

    // Enter IRQ ('10010') or Monitor ('10110') mode, and ensure Secure state if initially
    // in Monitor mode. This affects the banked versions of various registers accessed
    // later in the code.
    if CPSR.M == '10110' then SCR.NS = '0';
    CPSR.M = if trap_to_monitor then '10110' else '10010';

    // Write return information to registers, and make further CPSR changes: IRQs disabled,
    // other interrupts disabled if appropriate, IT state reset, instruction set and
    // endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';
    if trap_to_monitor then
        CPSR.F = '1';  CPSR.A = '1';
    else
        if !HaveSecurityExt() || SCR.NS == '0' || SCR.AW == '1' then CPSR.A = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1:Thumb
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
// Branch to correct IRQ vector.
if trap_to_monitor then
    BranchTo(MVBAR + 24);
elsif SCTLR.VE == '1' then
    IMPLEMENTATION_DEFINED branch to an IRQ vector;
else
    BranchTo(ExcVectorBase() + 24);
```

The preferred exception return from an IRQ interrupt is a return to the instruction following the instruction boundary at which the interrupt occurred. Use the LR and SPSR values generated by the exception entry to produce this return by using an exception return instruction with a subtraction of 4.

For more information, see *Exception return* on page B1-38.

B1.6.17 FIQ exception

The FIQ exception is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an FIQ interrupt request input to the processor.

Whether and how an FIQ exception is taken depends on the CPSR.F and SCTLR.FI bits:

- If CPSR.F == 1, FIQ exceptions are disabled and are not taken.
- If CPSR.F == 0 and SCTLR.FI == 0, FIQ exceptions can be taken. In this case FIQ exception entry is precise to an instruction boundary.
- If CPSR.F == 0 and SCTLR.FI == 1, FIQ exceptions can be taken. In this case FIQ exception entry is precise to an instruction boundary, except that some of the effects of the instruction that follows that boundary might have occurred. These effects are restricted to those that can be repeated idempotently and without breaking the rules in *Single-copy atomicity* on page A3-27. Examples of such effects are:
  - — changing the value of a register that the instruction writes but does not read
  - — performing an access to Normal memory.

  Note

  This relaxation of the normal definition of a precise asynchronous exception permits interrupts to occur during the execution of instructions that change register or memory values, while only requiring the implementation to restore those register values that are needed to correctly re-execute the instruction after the preferred exception return. `LDM` and `STM` are examples of such instructions.

The FIQ vector is the last vector in the vector table. This means the FIQ exception handler can be placed directly at the FIQ vector address, see *Exception vectors and the exception base address* on page B1-30. For example, if High vectors are enabled and VE == 0 the FIQ exception handler software can be placed at `0xFFFF001C`. This avoids a branch instruction from the vector.

The following pseudocode describes how this exception is taken:

```
// TakeFIQException()
// ==================
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

```
CPSR.E = SCTLR.EE;                // EE=0: little-endian, EE=1: big-endian

// Branch to correct IRQ vector.
if trap_to_monitor then
    BranchTo(MVBAR + 24);
elsif SCTLR.VE == '1' then
    IMPLEMENTATION_DEFINED branch to an IRQ vector;
else
    BranchTo(ExcVectorBase() + 24);
```

IRQ 割り込みからの望ましい例外復帰は、割り込みが発生した命令境界に続く命令に復帰することです。例外の開始によって生成された LR と SPSR の値と、4 の減算を伴う例外からの復帰命令を使用して、この復帰を実現します。

詳細については、*例外からの復帰* P. B1-38 を参照して下さい。

B1.6.17 FIQ 例外

FIQ 例外は、実装定義の手段によって生成されます。通常は、プロセッサへの FIQ 割り込み要求入力をアサートすることによって生成されます。

FIQ 例外を取得するか、どのように取得するかは、CPSR.F ビットと SCTLR.FI ビットに依存します。

- CPSR.F == 1 の場合は、FIQ 例外が禁止され、取得されません。
- CPSR.F == 0 および SCTLR.FI == 0 の場合は、FIQ 例外を取得することができます。この場合は、FIQ 例外の開始は正確に命令境界です。
- CPSR.F == 0 かつ SCTLR.FI == 1 の場合は、FIQ 例外を取得することができます。この場合は、FIQ 例外の開始は正確に命令境界ですが、その境界に従う命令の影響のいくつかが発生している可能性があります。この影響は、ベキ等に繰り返すことが可能で、*シングルコピーのアトミック性* P. A3-27 内の規則に違反しないものに限定されます。このような影響の例には、次のようなものがあります。
  - — 命令によって書き込まれるが読み出されないレジスタの値の変更
  - — ノーマルメモリへのアクセスの実行

  注

  正確な非同期例外の通常の定義をこのように緩和することで、レジスタまたはメモリの値を変更する命令の実行中に割り込みの発生が許可されます。実装で復元する必要があるのは、望ましい例外からの復帰の後に命令を正しく再実行するために必要なレジスタの値のみです。このような命令の例には、`LDM` および `STM` があります。

FIQ ベクタは、ベクタテーブルで最後のベクタです。これは、FIQ 例外ハンドラを直接 FIQ ベクタアドレスに配置できることを意味します。*例外ベクタと例外ベースアドレス* P. B1-30 を参照して下さい。例えば、ハイベクタが許可されていて、VE == 0 の場合は、FIQ 例外ハンドラソフトウェアを `0xFFFF001C` に配置することができます。これによって、ベクタからの分岐命令が不要になります。

次の擬似コードは、この例外の取得方法を示すものです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
TakeFIQException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the instruction boundary at which the interrupt occurred plus 4. For this
    // purpose, the PC and CPSR are considered to have already moved on to their values
    // for the instruction following that boundary.
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;

    // Determine whether FIQs are trapped to Monitor mode.
    trap_to_monitor = HaveSecurityExt() && SCR.FIQ == '1';

    // Enter FIQ ('10001') or Monitor ('10110') mode, and ensure Secure state if initially
    // in Monitor mode. This affects the banked versions of various registers accessed
    // later in the code.
    if CPSR.M == '10110' then SCR.NS = '0';
    CPSR.M = if trap_to_monitor then '10110' else '10001';

    // Write return information to registers, and make further CPSR changes: IRQs disabled,
    // other interrupts disabled if appropriate, IT state reset, instruction set and
    // endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';
    if trap_to_monitor then
        CPSR.F = '1';  CPSR.A = '1';
    else
        if !HaveSecurityExt() || SCR.NS == '0' || SCR.FW == '1' then CPSR.F = '1';
        if !HaveSecurityExt() || SCR.NS == '0' || SCR.AW == '1' then CPSR.A = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1: Thumb
    CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

    // Branch to correct FIQ vector.
    if trap_to_monitor then
        BranchTo(MVBAR + 28);
    elsif SCTLR.VE == '1' then
        IMPLEMENTATION_DEFINED branch to an FIQ vector;
    else
        BranchTo(ExcVectorBase() + 28);
```

The preferred exception return from an FIQ interrupt is a return to the instruction following the instruction boundary at which the interrupt occurred. Use the LR and SPSR values generated by the exception entry to produce this return by using an exception return instruction with a subtraction of 4.

For more information, see *Exception return* on page B1-38.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
// TakeFIQException()
// ==================

TakeFIQException()
    // Determine return information. SPSR is to be the current CPSR, and LR is to be the
    // current PC minus 0 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
    // respectively from the address of the current instruction into the required address
    // of the instruction boundary at which the interrupt occurred plus 4. For this
    // purpose, the PC and CPSR are considered to have already moved on to their values
    // for the instruction following that boundary.
    new_lr_value = if CPSR.T == '1' then PC else PC-4;
    new_spsr_value = CPSR;

    // Determine whether FIQs are trapped to Monitor mode.
    trap_to_monitor = HaveSecurityExt() && SCR.FIQ == '1';

    // Enter FIQ ('10001') or Monitor ('10110') mode, and ensure Secure state if initially
    // in Monitor mode. This affects the banked versions of various registers accessed
    // later in the code.
    if CPSR.M == '10110' then SCR.NS = '0';
    CPSR.M = if trap_to_monitor then '10110' else '10001';

    // Write return information to registers, and make further CPSR changes: IRQs disabled,
    // other interrupts disabled if appropriate, IT state reset, instruction set and
    // endianness to SCTLR-configured values.
    SPSR[] = new_spsr_value;
    R[14] = new_lr_value;
    CPSR.I = '1';
    if trap_to_monitor then
        CPSR.F = '1';  CPSR.A = '1';
    else
        if !HaveSecurityExt() || SCR.NS == '0' || SCR.FW == '1' then CPSR.F = '1';
        if !HaveSecurityExt() || SCR.NS == '0' || SCR.AW == '1' then CPSR.A = '1';
    CPSR.IT = '00000000';
    CPSR.J = '0';  CPSR.T = SCTLR.TE;  // TE=0: ARM, TE=1:Thumb
    CPSR.E = SCTLR.EE;                 // EE=0: little-endian, EE=1: big-endian

    // Branch to correct FIQ vector.
    if trap_to_monitor then
        BranchTo(MVBAR + 28);
    elsif SCTLR.VE == '1' then
        IMPLEMENTATION_DEFINED branch to an FIQ vector;
    else
        BranchTo(ExcVectorBase() + 28);
```

FIQ 割り込みからの望ましい例外復帰は、割り込みが発生した命令境界に続く命令に復帰することです。例外の開始によって生成された LR と SPSR の値と、4 の減算を伴う例外からの復帰命令を使用して、この復帰を実現します。

詳細については、*例外からの復帰* P. B1-38 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.7 Coprocessors and system control

The ARM architecture supports sixteen coprocessors, usually referred to as CP0 to CP15. These coprocessors are introduced in *Coprocessor support* on page A2-68. The architecture reserves two of these coprocessors, CP14 and CP15, for configuration and control related to the architecture:

- CP14 is reserved for the configuration and control of:
  - — debug features, see *The CP14 debug register interfaces* on page C6-32
  - — execution environment features, see *Execution environment support* on page B1-73.
- CP15 is called the System Control coprocessor, and is reserved for the control and configuration of the ARM processor system, including architecture and feature identification.

This section gives:

- general information about the CP15 registers, in *CP15 System Control coprocessor registers*
- information about access controls for coprocessors CP0 to CP13, in *Access controls on CP0 to CP13* on page B1-63.

B1.7.1 CP15 System Control coprocessor registers

The implementation of the CP15 registers depends heavily on whether the ARMv7 implementation is:

- an ARMv7-A implementation with a *Virtual Memory System Architecture* (VMSA)
- an ARMv7-R implementation with a *Protected Memory System Architecture* (PMSA).

Therefore, detailed descriptions of the CP15 registers are given in:

- *CP15 registers for a VMSA implementation* on page B3-64
- *CP15 registers for a PMSA implementation* on page B4-22.

Registers that are common to VMSA and PMSA implementations are described in both of these sections. Some registers are implemented differently in VMSA and PMSA implementations.

Those descriptions do not include the registers that implement the processor identification scheme, CPUID. The CPUID registers are described in Chapter B5 *The CPUID Identification Scheme*.

CP15, the System Control coprocessor, can contain up to 16 primary registers, each of which is 32 bits long. The CP15 register access instructions define the required primary register. Additional fields in the instruction are used to refine the access, and increase the number of physical 32-bit registers in CP15. In descriptions of the System Control coprocessor the 4-bit primary register number is used as a top level register identifier, because it is the primary factor determining the function of the register. The 16 primary registers in CP15 are identified as c0 to c15.

For details of register access rights and restrictions see the descriptions of the individual registers. In ARMv7-A implementations, see also *Effect of the Security Extensions on the CP15 registers* on page B3-71.

The CP15 register access instructions are:

- `MCR`, to write an ARM core register to a CP15 register, see *MCR, MCR2* on page A8-186
- `MRC`, to read the value of a CP15 register into an ARM core register, see *MRC, MRC2* on page A8-202.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B1.7 コプロセッサとシステム制御

ARM アーキテクチャでは、通常は CP0 ～ CP15 として参照される 16 個のコプロセッサがサポートされます。これらのコプロセッサについては、*コプロセッサのサポート* P. A2-68 で説明します。アーキテクチャでは、これらのコプロセッサのうち CP14 と CP15 の 2 つが、アーキテクチャに関連した構成と制御用に予約されています。

- CP14 は、次の構成と制御用に予約されています。
  - – デバッグ機能（*CP14 デバッグレジスタ インタフェース* P. C6-32 を参照）
  - – 実行環境機能（*実行環境のサポート* P. B1-73 を参照）
- CP15 はシステム制御コプロセッサと呼ばれており、アーキテクチャと機能の識別を含む ARM プロセッサシステムの制御と構成用に予約されています。

このセクションでは、次の内容について説明します。

- CP15 レジスタに関する一般的な情報（*CP15、システム制御コプロセッサレジスタ*）
- コプロセッサの CP0 ～ CP13 のアクセス制御に関する情報（*CP0 ～ CP13 のアクセス制御* P. B1-63）

B1.7.1 CP15、システム制御コプロセッサレジスタ

CP15 レジスタの実装は、ARMv7 実装が次のどちらかによって大きく異なります。

- *仮想メモリシステム アーキテクチャ*（VMSA）に基づく ARMv7-A 実装
- *保護メモリシステム アーキテクチャ*（PMSA）に基づく ARMv7-R 実装

したがって、CP15 レジスタの詳細な説明については、次のセクションを参照して下さい。

- *VMSA 実装の CP15 レジスタ* P. B3-64
- *PMSA 実装の CP15 レジスタ* P. B4-22

VMSA 実装と PMSA 実装に共通のレジスタについては、両方のセクションで説明します。VMSA 実装と PMSA 実装では、一部のレジスタの実装方法が異なります。

これらの説明には、プロセッサ識別方式の CPUID を実装するレジスタが含まれていません。CPUID レジスタについては、第 B5 章*「CPUID 識別方式」*で説明します。

システム制御コプロセッサ（CP15）には、最大で 16 個の 1 次レジスタ（それぞれが 32 ビット長）を含めることができます。CP15 レジスタアクセス命令によって、必要な 1 次レジスタが定義されます。命令内の他のフィールドは、アクセスを細分して、CP15 内の物理的な 32 ビットレジスタの数を増やすために使用されます。システム制御コプロセッサの説明では、4 ビットの 1 次レジスタ番号が最上位のレジスタ識別子として使用されます。これは、この番号がレジスタの機能を決定する主要素であるためです。CP15 内の 16 個の 1 次レジスタは、c0 ～ c15 として識別されます。

レジスタのアクセス権限と制限の詳細については、個別のレジスタの説明を参照して下さい。ARMv7-A 実装の場合は、*CP15 レジスタに対するセキュリティ拡張機能の影響* P. B3-71 も参照して下さい。

CP15 レジスタアクセス命令は次のとおりです。

- ARM コアレジスタを CP15 レジスタに書き込むための `MCR`（*MCR*、*MCR2* P. A8-186 を参照）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

All CP15 CDP, CDP2, LDC, LDC2, MCR2, MCRR, MCRR2, MRC2, MRRC, MRRC2, STC, and STC2 instructions are UNDEFINED.

B1.7.2 Access controls on CP0 to CP13

Coprocessors CP0 to CP13 might be required for optional features of the ARMv7 implementation. In particular, CP10 and CP11 are used to support floating-point operations through the VFP and Advanced SIMD extensions to the architecture, see *Advanced SIMD and floating-point support* on page B1-64.

Coprocessors CP0 to CP7 can be used to provide IMPLEMENTATION DEFINED vendor specific features.

Access to the coprocessors CP0 to CP13 is controlled by the Coprocessor Access Control Register, see:

- *c1, Coprocessor Access Control Register (CPACR)* on page B3-104 for a VMSA implementation
- *c1, Coprocessor Access Control Register (CPACR)* on page B4-51 for a PMSA implementation.

Initially on power up or reset, access to coprocessors CP0 to CP13 is disabled.

When the Security Extensions are implemented, the Non-Secure Access Control Register determines which of the CP0 to CP13 coprocessors can be accessed from the Non-secure state, see *c1, Non-Secure Access Control Register (NSACR)* on page B3-110.

Note

- When an implementation includes either or both of the VFP and Advanced SIMD extensions, the access settings for CP10 and CP11 must be identical. If these settings are not identical the behavior of the extensions is UNPREDICTABLE.
- To check which coprocessors are implemented:
  1. If required, read the Coprocessor Access Control Register and save the value.
  2. Write the value 0x0FFFFFFF to the register, to write 0b11 to the access field for each of the coprocessors CP13 to CP0.
  3. Read the Coprocessor Access Control Register again and check the access field for each coprocessor:
     - if the access field value is 0b00 the coprocessor is not implemented
     - if the access field value is 0b11 the coprocessor is implemented.
  4. If required, write the value from stage 1 back to the register to restore the original value.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- CP15 レジスタの値を ARM コアレジスタに読み出すための MRC（*MRC*、*MRC2* P. A8-202 を参照）

CP15 の CDP、CDP2、LDC、LDC2、MCR2、MCRR、MCRR2、MRC2、MRRC、MRRC2、STC、STC2 の各命令はすべて未定義です。

B1.7.2 CP0 ～ CP13 のアクセス制御

コプロセッサの CP0 ～ CP13 は、ARMv7 実装のオプション機能で必要になる場合があります。特に、CP10 と CP11 は、アーキテクチャの VFP 拡張機能とアドバンスト SIMD 拡張機能によって浮動小数点演算をサポートするために使用されます。*アドバンスト SIMD と浮動小数点のサポート* P. B1-64 を参照して下さい。

コプロセッサの CP0 ～ CP7 は、実装定義のベンダ固有の機能を提供するために使用できます。

コプロセッサ CP0 ～ CP13 へのアクセスは、コプロセッサアクセス制御レジスタによって制御されます。次のセクションを参照して下さい。

- *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B3-104（VMSA 実装の場合）
- *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B4-51 （PMSA 実装の場合）

初期の起動時またはリセット時は、コプロセッサ CP0 ～ CP13 へのアクセスが禁止されます。

セキュリティ拡張機能が実装されている場合は、非セキュアアクセス制御レジスタによって、非セキュア状態から CP0 ～ CP13 のどのコプロセッサにアクセスできるかが決定されます。*c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3-110 を参照して下さい。

注

- 実装に VFP 拡張機能とアドバンスト SIMD 拡張機能のいずれかまたは両方が含まれている場合は、CP10 と CP11 のアクセス設定を同じにする必要があります。この設定が異なる場合は、拡張機能の動作が予測不能になります。
- 実装されているコプロセッサをチェックするには、次の手順を使用します。
  1. 必要に応じて、コプロセッサアクセス制御レジスタを読み出してその値を保存します。
  2. 0x0FFFFFFF の値をこのレジスタに書き込んで、コプロセッサ CP0 ～ CP13 のそれぞれのアクセスフィールドに 0b11 を書き込みます。
  3. 再度コプロセッサアクセス制御レジスタを読み出して、各コプロセッサのアクセスフィールドをチェックします。
     - アクセスフィールドの値が 0b00 の場合は、コプロセッサが実装されていません。
     - アクセスフィールドの値が 0b11 の場合は、コプロセッサが実装されています。
  4. 必要に応じて、手順 1 からの値をレジスタに戻して元の値を復元します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.8 Advanced SIMD and floating-point support

Advanced SIMD and VFP extensions on page A2-20 introduces:

- the VFP extension, used for scalar floating-point operations
- the Advanced SIMD extension, used for integer and floating-point vector operations
- the Advanced SIMD and VFP extension registers D0 - D31 and their alternative views as S0 - S31 and Q0 - Q15
- the *Floating-Point Status and Control Register* (FPSCR).

For more information about the system registers for the Advanced SIMD and VFP extensions see *Advanced SIMD and VFP extension system registers* on page B1-66.

Software can interrogate the registers described in *Advanced SIMD and VFP feature identification registers* on page B5-34 to discover the Advanced SIMD and floating-point support implemented in a system.

This section gives more information about the Advanced SIMD and VFP extensions, in the subsections:

- *Enabling Advanced SIMD and floating-point support*
- *Advanced SIMD and VFP extension system registers* on page B1-66
- *The Floating-Point Exception Register (FPEXC)* on page B1-68
- *Context switching with the Advanced SIMD and VFP extensions* on page B1-69
- *VFP support code* on page B1-70
- *VFP subarchitecture support* on page B1-72.

B1.8.1 Enabling Advanced SIMD and floating-point support

If an ARMv7 implementation includes support for any Advanced SIMD or VFP features then the boot software for any system that uses that implementation must ensure that:

- access to CP10 and CP11 is enabled in the Coprocessor Access Control Register, see:
  - *c1, Coprocessor Access Control Register (CPACR)* on page B3-104 for a VMSA implementation
  - *c1, Coprocessor Access Control Register (CPACR)* on page B4-51 for a PMSA implementation.
- if the Security Extensions are implemented and Non-secure access to the Advanced SIMD or VFP features is required, the access flags for CP10 and CP11 in the NSACR must be set to 1, see *c1, Non-Secure Access Control Register (NSACR)* on page B3-110.

If this is not done, operation of Advanced SIMD and VFP features is UNDEFINED.

If the access control bits are programmed differently for CP10 and CP11, operation of Advanced SIMD and VFP features is UNPREDICTABLE.

In addition, software must set the FPEXC.EN bit to 1 to enable most Advanced SIMD and VFP operations, see *The Floating-Point Exception Register (FPEXC)* on page B1-68.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B1.8 アドバンスト SIMD と浮動小数点のサポート

アドバンスト SIMD および VFP 拡張機能 P. A2-20 では、次の内容について説明します。

- スカラ浮動小数点演算で使用される VFP 拡張機能
- 整数演算と浮動小数点ベクタ演算で使用されるアドバンスト SIMD 拡張機能
- アドバンスト SIMD および VFP 拡張機能のレジスタ D0 ～ D31 と、それらのレジスタに対する S0 ～ S31 および Q0 ～ Q15 としてのビュー
- *浮動小数点ステータスおよび制御レジスタ*(FPSCR)

アドバンスト SIMD 拡張機能と VFP 拡張機能のシステムレジスタの詳細については、*アドバンスト SIMD および VFP 拡張機能のシステムレジスタ* P. B1-66 を参照して下さい。

アドバンスト SIMD および VFP 機能識別レジスタ P. B5-34 で説明されているレジスタをソフトウェアで調査して、システムに実装されているアドバンスト SIMD および浮動小数点のサポートを検出できます。

このセクションでは、次のサブセクションで、アドバンスト SIMD 拡張機能と VFP 拡張機能について詳しく説明します。

- *アドバンスト SIMD と浮動小数点のサポートの許可*
- *アドバンスト SIMD および VFP 拡張機能のシステムレジスタ* P. B1-66
- *浮動小数点例外レジスタ (FPEXC)* P. B1-68
- *アドバンスト SIMD 拡張機能と VFP 拡張機能によるコンテキストスイッチ* P. B1-69
- *VFP サポートコード* P. B1-70
- *VFP サブアーキテクチャのサポート* P. B1-72

B1.8.1 アドバンスト SIMD と浮動小数点のサポートの許可

ARMv7 実装にアドバンスト SIMD 機能または VFP 機能に対するサポートが含まれている場合は、その実装を使用するすべてのシステムのブートソフトウェアで、次のことを保証する必要があります。

- CP10 と CP11 へのアクセスが、コプロセッサアクセス制御レジスタで許可されている。次のセクションを参照して下さい。
  - *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B3-104（VMSA 実装の場合）
  - *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B4-51 （PMSA 実装の場合）
- セキュリティ拡張機能が実装されており、アドバンスト SIMD 機能または VFP 機能への非セキュアアクセスが必要な場合は、NSACR 内の CP10 と CP11 のアクセスフラグを 1 に設定する必要がある。*c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3-110 を参照して下さい。

これが保証されない場合は、アドバンスト SIMD 機能と VFP 機能の動作が未定義になります。

CP10 と CP11 のアクセス制御ビットが異なる設定にプログラムされると、アドバンスト SIMD 機能と VFP 機能の動作が予測不能になります。

さらに、ほとんどのアドバンスト SIMD および VFP 動作は、ソフトウェアで FPEXC.EN ビットを 1 にセットして許可する必要があります。*浮動小数点例外レジスタ (FPEXC)* P. B1-68 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When floating-point operation is disabled because FPEXC.EN is 0, all Advanced SIMD and VFP instructions are treated as Undefined Instructions except for:

- a VMSR to the FPEXC or FPSID register
- a VMRS from the FPEXC, FPSID, MVFR0, or MVFR1 register.

These instructions can be executed only in privileged modes.

Note

- When FPEXC.EN == 0, these operations are treated as Undefined Instructions:
  - — a VMSR to the FPSCR
  - — a VMRS from the FPSCR
- If a VFP implementation contains system registers additional to the FPSID, FPSCR, FPEXC, MVFR0, and MVFR1 registers, the behavior of VMSR instructions to them and VMRS instructions from them is SUBARCHITECTURE DEFINED.

Pseudocode details of enabling the Advanced SIMD and VFP extensions

The following pseudocode takes appropriate action if an Advanced SIMD or VFP instruction is used when the extensions are not enabled:

```
// CheckAdvSIMDOrVFPEnabled()
// ==========================

CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
    if HaveSecurityExt() then
        // Check Non-secure Access Control Register for permission to use CP10/11.
        if NSACR.cp10 != NSACR.cp11 then UNPREDICTABLE;
        if SCR.NS == '1' && NSACR.cp10 == '0' then UNDEFINED;

    // Check Coprocessor Access Control Register for permission to use CP10/11.
    if CPACR.cp10 != CPACR.cp11 then UNPREDICTABLE;
    case CPACR.cp10 of
        when '00'  UNDEFINED;
        when '01'  if !CurrentModeIsPrivileged() then UNDEFINED; // else CPACR permits access
        when '10'  UNPREDICTABLE;
        when '11'  // CPACR permits access

    // If the Advanced SIMD extension is specified, check whether it is disabled.
    if advsimd && CPACR.ASEDIS == '1' then UNDEFINED;

    // If required, check FPEXC enabled bit.
    if include_fpexc_check && FPEXC.EN == '0' then UNDEFINED;

    return;

// CheckAdvSIMDEnabled()
// =====================

CheckAdvSIMDEnabled()
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

FPEXC.EN が 0 のために浮動小数点演算が禁止されている場合は、次の命令を除いて、すべてのアドバンスト SIMD 命令および VFP 命令が未定義命令として扱われます。

- FPEXC または FPSID レジスタに対する VMSR
- FPEXC、FPSID、MVFR0、または MVFR1 レジスタからの VMRS

これらの命令は、特権モードでのみ実行できます。

注

- FPEXC.EN == 0 の場合は、次の動作が未定義命令として扱われます。
  - – FPSCR に対する VMSR
  - – FPSCR からの VMRS
- VFP 実装に FPSID、FPSCR、FPEXC、MVFR0、MVFR1 レジスタ以外のシステムレジスタが含まれている場合、それらに対する VMSR 命令と VMRS 命令の動作はサブアーキテクチャ定義です。

アドバンスト SIMD 拡張機能と VFP 拡張機能を許可する擬似コードの詳細

次の擬似コードは、拡張機能が許可されていない状態でアドバンスト SIMD 命令または VFP 命令が使用された場合に、適切な動作を実行します。

```
// CheckAdvSIMDOrVFPEnabled()
// ==========================

CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
    if HaveSecurityExt() then
        // Check Non-secure Access Control Register for permission to use CP10/11.
        if NSACR.cp10 != NSACR.cp11 then UNPREDICTABLE;
        if SCR.NS == '1' && NSACR.cp10 == '0' then UNDEFINED;

    // Check Coprocessor Access Control Register for permission to use CP10/11.
    if CPACR.cp10 != CPACR.cp11 then UNPREDICTABLE;
    case CPACR.cp10 of
        when '00'  UNDEFINED;
        when '01'  if !CurrentModeIsPrivileged() then UNDEFINED; // else CPACR permits access
        when '10'  UNPREDICTABLE;
        when '11'  // CPACR permits access

    // If the Advanced SIMD extension is specified, check whether it is disabled.
    if advsimd && CPACR.ASEDIS == '1' then UNDEFINED;

    // If required, check FPEXC enabled bit.
    if include_fpexc_check && FPEXC.EN == '0' then UNDEFINED;

    return;

// CheckAdvSIMDEnabled()
// =====================

CheckAdvSIMDEnabled()
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    return CheckAdvSIMDOrVFPEnabled(TRUE, TRUE);

// CheckVFPEnabled()
// =================

CheckVFPEnabled(boolean include_fpexc_check)
    return CheckAdvSIMDOrVFPEnabled(include_fpexc_check, FALSE);
```

B1.8.2 Advanced SIMD and VFP extension system registers

The Advanced SIMD and VFP extensions share a common set of special-purpose registers. Any ARMv7 implementation that includes either or both of these extensions must implement these registers. This section gives general information about this set of registers, and indicates where each register is described in detail. It contains the following subsections:

- *Register map of the Advanced SIMD and VFP extension system registers*
- *Accessing the Advanced SIMD and VFP extension system registers* on page B1-67.

Register map of the Advanced SIMD and VFP extension system registers

Table B1-11 shows the register map of the Advanced SIMD and VFP registers. When the Security Extensions are implemented, the Advanced SIMD and VFP registers are not banked.

Table B1-11 Advanced SIMD and VFP common register block

System register	Name	Description
0b0000	FPSID	See *Floating-point System ID Register (FPSID)* on page B5-34
0b0001	FPSCR	See *Floating-point Status and Control Register (FPSCR)* on page A2-28
0b0010- 0b0101	Reserved	All accesses are UNPREDICTABLE
0b0110	MVFR1	Media and VFP Feature Registers 1 and 0, see *Media and VFP Feature registers* on page B5-36
0b0111	MVFR0	
0b1000	FPEXC	See *The Floating-Point Exception Register (FPEXC)* on page B1-68
0b1001-0b1111	SUBARCHITECTURE DEFINED	-

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳


```
    return CheckAdvSIMDOrVFPEnabled(TRUE, TRUE);

// CheckVFPEnabled()
// =================

CheckVFPEnabled(boolean include_fpexc_check)
    return CheckAdvSIMDOrVFPEnabled(include_fpexc_check, FALSE);
```


B1.8.2 アドバンスト SIMD および VFP 拡張機能のシステムレジスタ

アドバンスト SIMD 拡張機能と VFP 拡張機能では、特殊レジスタの共通セットが共有されます。これらの拡張機能のいずれかまたは両方を含むすべての ARMv7 実装で、これらのレジスタを実装する必要があります。このセクションでは、このレジスタのセットに関する一般的な情報を提供し、各レジスタの詳細が記載された場所を示します。このセクションは、次のサブセクションで構成されています。

- *アドバンスト SIMD および VFP 拡張機能システムレジスタのレジスタマップ*
- *アドバンスト SIMD および VFP 拡張機能システムレジスタへのアクセス* P. B1-67

アドバンスト SIMD および VFP 拡張機能システムレジスタのレジスタマップ

アドバンスト SIMD レジスタと VFP レジスタのレジスタマップを、表 B1-11 に示します。セキュリティ拡張機能が実装されている場合は、アドバンスト SIMD レジスタと VFP レジスタがバンクされません。

表 B1-11 アドバンスト SIMD と VFP に共通のレジスタブロック

システムレジスタ	名前	説明
0b0000	FPSID	*浮動小数点システム ID レジスタ (FPSID)* P. B5-34 を参照
0b0001	FPSCR	*浮動小数点ステータスおよび制御レジスタ (FPSCR)* P. A2-28 を参照
0b0010 ～ 0b0101	予約	すべてのアクセスが予測不能です。
0b0110	MVFR1	メディア / VFP 機能レジスタ 0/1（*メディアおよび VFP 機能レジスタ* P. B5-36 を参照）
0b0111	MVFR0	
0b1000	FPEXC	*浮動小数点例外レジスタ (FPEXC)* P. B1-68 を参照
0b1001 ～ 0b1111	サブアーキテクチャ定義	–

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

- Appendix B *Common VFP Subarchitecture Specification* includes examples of how a VFP subarchitecture might define additional registers, in the SUBARCHITECTURE DEFINED register space using addresses in the 0b1001 to 0b1111 range.
- Appendix B is not part of the ARMv7 architecture. It is included as an example of how a VFP subarchitecture might be defined.

Accessing the Advanced SIMD and VFP extension system registers

You access the Advanced SIMD and VFP extension system registers using the `VMRS` and `VMSR` instructions, see:

- *VMRS* on page A8-658
- *VMSR* on page A8-660.

For example:

```
VMRS <Rt>, FPSID    ; Read  Floating-Point System ID Register
VMRS <Rt>, MVFR1    ; Read  Media and VFP Feature Register 1
VMSR FPSCR, <Rt>    ; Write Floating-Point System Control Register
```

You must enable access to CP10 and CP11 in the Coprocessor Access Control register before you can access any of the Advanced SIMD and VFP extension system registers, see *Enabling Advanced SIMD and floating-point support* on page B1-64.

To enable access to the FPSCR you must also set the EN flag in the FPEXC Register to 1, see *The Floating-Point Exception Register (FPEXC)* on page B1-68.

Table B1-12 shows the permitted accesses to the Advanced SIMD and VFP extension system registers when the access rights to CP10 and CP11 are sufficient.

Table B1-12 Access to Advanced SIMD and VFP system registers

Register	Register access	Privileged accesses		User accesses	
		EN == 0 [a]	EN == 1 [a]	EN == 0 [a]	EN == 1 [a]
FPSID	Read-only	Permitted	Permitted	Not permitted	Not permitted
FPSCR	Read/write	Not permitted	Permitted	Not permitted	Permitted
MVFR1, MVFR0	Read-only	Permitted	Permitted	Not permitted	Not permitted
FPEXC	Read/write	Permitted	Permitted	Not permitted	Not permitted

a. In the FPEXC Register, see *The Floating-Point Exception Register (FPEXC)* on page B1-68.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

- VFP サブアーキテクチャで 0b1001 ～ 0b1111 の範囲のアドレスを使用して、サブアーキテクチャ定義のレジスタ空間に追加のレジスタを定義する場合の例を、付録 B *「一般的な VFP サブアーキテクチャ仕様」*に示します。
- 付録 B は、ARMv7 アーキテクチャの一部ではありません。この付録は、VFP サブアーキテクチャの定義方法を示す例として含まれています。

アドバンスト SIMD および VFP 拡張機能システムレジスタへのアクセス

アドバンスト SIMD および VFP 拡張機能システムレジスタには、`VMRS` 命令と `VMSR` 命令を使用してアクセスします。次のセクションを参照して下さい。

- *VMRS* P. A8- 658
- *VMSR* P. A8- 660

次に例を示します。

```
VMRS <Rt>, FPSID    ; Read  Floating-Point System ID Register
VMRS <Rt>, MVFR1    ; Read  Media and VFP Feature Register 1
VMSR FPSCR, <Rt>    ; Write Floating-Point System Control Register
```

アドバンスト SIMD および VFP 拡張機能システムレジスタにアクセスするには、コプロセッサアクセス制御レジスタで CP10 と CP11 へのアクセスを許可する必要があります。*アドバンスト SIMD と浮動小数点のサポートの許可* P. B1- 64 を参照して下さい。

FPSCR へのアクセスを許可するには、FPEXC レジスタ内の EN フラグを 1 にセットすることも必要です。*浮動小数点例外レジスタ (FPEXC)* P. B1- 68 を参照して下さい。

CP10 と CP11 へのアクセス権が十分な場合、アドバンスト SIMD および VFP 拡張機能システムレジスタに対して許可されるアクセスを、表 B1- 12 に示します。

表 B1- 12 アドバンスト SIMD システムレジスタと VFP システムレジスタへのアクセス

レジスタ	レジスタアクセス	特権アクセス		ユーザアクセス	
		EN ＝ 0 [a]	EN ＝ 1 [a]	EN ＝ 0 [a]	EN ＝ 1 [a]
FPSID	読み出し専用	許可	許可	禁止	禁止
FPSCR	読み出し / 書き込み	禁止	許可	禁止	許可
MVFR1、MVFR0	読み出し専用	許可	許可	禁止	禁止
FPEXC	読み出し / 書き込み	許可	許可	禁止	禁止

a. FPEXC レジスタの場合は、*浮動小数点例外レジスタ (FPEXC)* P. B1- 68 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

All hardware ID information can be accessed only from privileged modes.

The FPSID is privileged access only.

This is a change in VFPv3. In VFPv2 implementations the FPSID register can be accessed in all modes.

The MVFR registers are privileged access only.

User code must issue a system call to determine what features are supported.

B1.8.3 The Floating-Point Exception Register (FPEXC)

The Floating-Point Exception Register (FPEXC) provides global enable and disable control of the Advanced SIMD and VFP extensions, and to indicate how the state of these extensions is recorded.

The FPEXC:

- Is in the CP10 and CP11 register space.
- Is present only when at least one of the VFP and Advanced SIMD extensions is implemented.
- Is a 32-bit read/write register, that can have different access rights for different bits.
- If the Security Extensions are implemented, is a Configurable access register. The FPEXC is only accessible in the Non-secure state if the CP10 and CP11 bits in the NSACR are set to 1, see *c1, Non-Secure Access Control Register (NSACR)* on page B3-110
- Is accessible only in privileged modes, and only if access to coprocessors CP10 and CP11 is enabled in the Coprocessor Access Control Register, see:
  — *c1, Coprocessor Access Control Register (CPACR)* on page B3-104 for a VMSA implementation
  — *c1, Coprocessor Access Control Register (CPACR)* on page B4-51 for a PMSA implementation.
- Has a reset value of 0 for bit [30], FPEXC.EN.

The format of the FPEXC is:

31	30	29 … 0
EX	EN	SUBARCHITECTURE DEFINED

EX, bit [31] Exception bit. A status bit that specifies how much information must be saved to record the state of the Advanced SIMD and VFP system:

0 The only significant state is the contents of the registers:
- D0 - D15
- D16 - D31, if implemented

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

すべてのハードウェア ID 情報には、特権モードでのみアクセスできます。

FPSID は特権アクセスのみ許可されます。

これは、VFPv3 で変更されたものです。VFPv2 実装では、FPSID レジスタにすべてのモードでアクセスすることができます。

MVFR レジスタは特権アクセスのみ許可されます。

ユーザコードはシステムコールを発行して、どの機能がサポートされているかを判断する必要があります。

B1.8.3 浮動小数点例外レジスタ (FPEXC)

浮動小数点例外レジスタ (FPEXC) では、アドバンスト SIMD 拡張機能と VFP 拡張機能のグローバルなイネーブル / ディセーブル制御が提供され、これらの拡張機能の状態がどのように記録されるかが示されます。

FPEXC には次の特徴があります。

- CP10 および CP11 のレジスタ空間に存在します。
- VFP 拡張機能とアドバンスト SIMD 拡張機能のいずれかまたは両方が実装されている場合にのみ存在します。
- 32 ビットの読み出し/書き込みレジスタで、ビットごとにアクセス権が異なる可能性があります。
- セキュリティ拡張機能が実装されている場合、構成可能なアクセスレジスタです。FPEXC には、NSACR の CP10 ビットと CP11 ビットが 1 に設定されている場合の非セキュア状態でのみアクセスできます。*c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3-110 を参照して下さい。
- 特権モードで、コプロセッサの CP10 と CP11 へのアクセスがコプロセッサアクセス制御レジスタ内で許可されている場合にのみ、アクセスできます。次のセクションを参照して下さい。
  – *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B3-104（VMSA 実装の場合）
  – *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B4-51 （PMSA 実装の場合）
- ビット [30](FPEXC.EN) のリセット時の値は 0 です。

FPEXC のフォーマットは次のとおりです。

31	30	29 … 0
EX	EN	サブアーキテクチャ定義

EX、ビット [31] 例外ビット。アドバンスト SIMD システムと VFP システムの状態を記録するため、どの程度の情報を保存する必要があるかを示すステータスビットです。

0 次のレジスタの内容のみが必要です。
- D0 ～ D15

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- FPSCR
- FPEXC.

A context switch can be performed by saving and restoring the values of these registers.

1 There is additional state that must be handled by any context switch system.

The behavior of the EX bit on writes is SUBARCHITECTURE DEFINED, except that in any implementation a write of 0 to this bit must be a valid operation, and must return a value of 0 if read back immediately.

EN, bit [30] Enable bit. A global enable for the Advanced SIMD and VFP extensions:

0 The Advanced SIMD and VFP extensions are disabled. For details of how the system operates when EN == 0 see *Enabling Advanced SIMD and floating-point support* on page B1-64.

1 The Advanced SIMD and VFP extensions are enabled and operate normally.

This bit is always a normal read/write bit. It has a reset value of 0.

Bits [29:0] SUBARCHITECTURE DEFINED. An implementation can use these bits to communicate exception information between the floating-point hardware and the support code. The subarchitectural definition of these bits includes their read/write access. This can be defined on a bit by bit basis.

A constraint on these bits is that if EX == 0 it must be possible to save and restore all significant state for the floating-point system by saving and restoring only the two Advanced SIMD and VFP extension registers FPSCR and FPEXC.

Writes to the FPEXC can have side-effects on various aspects of processor operation. All of these side-effects are synchronous to the FPEXC write. This means they are guaranteed not to be visible to earlier instructions in the execution stream, and they are guaranteed to be visible to later instructions in the execution stream.

See *Advanced SIMD and VFP extension system registers* on page B1-66 for an overview of the common set of system registers for the Advanced SIMD and VFP extensions.

B1.8.4 Context switching with the Advanced SIMD and VFP extensions

In an implementation that includes one or both of the Advanced SIMD and VFP extensions, if the VFP registers are used by only a subset of processes, the operating system might implement lazy context switching of the extension registers and extension system registers.

In the simplest lazy context switch implementation, the primary context switch code simply disables the VFP and Advanced SIMD extensions, by disabling access to coprocessors CP10 and CP11 in the Coprocessor Access Control Register, see *Enabling Advanced SIMD and floating-point support* on page B1-64. Subsequently, when a process or thread attempts to use an Advanced SIMD or VFP instruction, it triggers an Undefined Instruction exception. The operating system responds by saving and restoring the extension registers and extension system registers.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- D16 ～ D31（実装されている場合）
- FPSCR
- FPEXC

これらのレジスタの値を保存して復元することによって、コンテキストスイッチを実行できます。

1 すべてのコンテキストスイッチ システムで処理する必要のある、他の状態が存在します。

書き込み時の EX ビットの動作は、サブアーキテクチャ定義です。ただし例外として、どのような実装でもこのビットに 0 を書き込む操作は有効な必要があり、その直後に読み出された場合は値 0 が返される必要があります。

EN、ビット [30] イネーブルビット。アドバンスト SIMD 拡張機能と VFP 拡張機能をグローバルに稼動 / 非稼動にします。

0 アドバンスト SIMD 拡張機能と VFP 拡張機能が禁止されます。EN == 0 の場合のシステム動作の詳細については、*アドバンスト SIMD と浮動小数点のサポートの許可* P. B1-64 を参照して下さい。

1 アドバンスト SIMD 拡張機能と VFP 拡張機能が許可されていて、正常に動作します。

このビットは常に、通常の読み出し / 書き込みビットです。リセット時の値は 0 です。

ビット [29:0] サブアーキテクチャ定義。実装でこれらのビットを使用して、浮動小数点ハードウェアとサポートコードとの間で例外情報をやり取りすることができます。これらのビットのサブアーキテクチャ定義には、その読み出し / 書き込みアクセスが含まれます。これは、ビット単位で定義することができます。

これらのビットに関する制約は、EX == 0 の場合、アドバンスト SIMD および VFP 拡張機能のレジスタ FPSCR と FPEXC の 2 つだけを保存 / 復元して、浮動小数点システムのすべての有意状態を保存して復元できなければならないことです。

FPEXC への書き込みは、プロセッサ動作のさまざまな側面に副作用をもたらす可能性があります。これらの副作用のすべては、FPEXC 書き込みと同期します。つまり、これらの副作用は、実行ストリーム内でその命令よりも前にある命令からは可視でなく、その命令より後にある命令からは可視であることが保証されます。

アドバンスト SIMD 拡張機能と VFP 拡張機能のシステムレジスタの共通セットの概要については、*アドバンスト SIMD および VFP 拡張機能のシステムレジスタ* P. B1-66 を参照して下さい。

B1.8.4 アドバンスト SIMD 拡張機能と VFP 拡張機能によるコンテキストスイッチ

アドバンスト SIMD 拡張機能と VFP 拡張機能のいずれかまたは両方を含む実装で、VFP レジスタがプロセスのサブセットでのみ使用されている場合は、オペレーティングシステムに拡張機能レジスタと拡張機能システムレジスタの低速コンテキストスイッチが実装される可能性があります。

最も単純な低速コンテキストスイッチ実装では、基本的なコンテキストスイッチ コードを使用して、コプロセッサアクセス制御レジスタでコプロセッサの CP10 と CP11 へのアクセスを禁止し、VFP 拡張機能とアドバンスト SIMD 拡張機能を禁止します。*アドバンスト SIMD と浮動小数点のサポートの許可* P. B1-64 を参照して下さい。その後で、プロセスまたはスレッドでアドバンスト SIMD 命令または VFP 命令を使用しようとしたときに、未定義命令例外がトリガされます。オペレーティングシステムは、拡張機能レジスタと拡張機能システムレジスタを保存して復元することによって応答します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.8.5 VFP support code

A complete VFP implementation might require a software component, known as the *support code*. For example, if VFPv3U is implemented support code must handle the trapped floating-point exceptions.

Typically, the support code is entered through the ARM Undefined Instruction vector, when the extension hardware does not respond to a VFP instruction. This software entry is known as a *bounce*.

When VFPv3U is implemented, the bounce mechanism is used to support trapped floating-point exceptions. Trapped floating-point exceptions, known as *traps,* are floating-point exceptions that an implementation passes back to application software to resolve, see *Floating-point exceptions* on page A2-42. The support code must catch a trapped exception and convert it into a trap handler call.

The support code can perform other tasks, as determined by the implementation. It might be used for rare conditions, such as operations that are difficult to implement in hardware, or operations that are gate intensive in hardware. This permits consistent software behavior with varying degrees of hardware support.

The division of labor between the hardware and software components of an implementation, and details of the interface between the support code and hardware are SUBARCHITECTURE DEFINED.

Asynchronous bounces, serialization, and VFP exception barriers

A VFP implementation can produce an *asynchronous bounce*, in which a VFP instruction takes the Undefined Instruction exception because support code processing is required for an earlier VFP instruction. The mechanism by which the nature of the required processing is communicated to the support code is SUBARCHITECTURE DEFINED. Typically, it involves:

- using the SUBARCHITECTURE DEFINED bits of the FPEXC, see *The Floating-Point Exception Register (FPEXC)* on page B1-68
- using the SUBARCHITECTURE DEFINED extension system registers, see *Advanced SIMD and VFP extension system registers* on page B1-66
- setting FPEXC.EX == 1, to indicate that the SUBARCHITECTURE DEFINED extension system registers must be saved on a context switch.

An asynchronous bounce might not relate to the last VFP instruction executed before the one that took the Undefined Instruction exception. It is possible that another VFP instruction has been issued and retired before the asynchronous bounce occurs. This is possible only if this intervening instruction has no register dependencies on the VFP instruction that requires support code processing. In addition. it is possible that there are SUBARCHITECTURE DEFINED mechanisms for handling an intervening VFP instruction that has issued but not retired.

However, `VMRS` and `VMSR` instructions that access the FPSID, FPSCR, or FPEXC registers are *serializing* instructions. This means they ensure that any exceptional condition in any preceding VFP instruction that requires support code processing has been detected and reflected in the extension system registers before they perform the register transfer. A `VMSR` instruction to the read-only FPSID register is a serializing NOP.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B1.8.5 VFP サポートコード

完全な VFP 実装には、*サポートコード*と呼ばれるソフトウェアコンポーネントが必要です。例えば、VFPv3U が実装されている場合は、トラップされた浮動小数点例外をサポートコードで処理する必要があります。

通常、サポートコードは、拡張機能ハードウェアが VFP 命令に応答しない場合の ARM 未定義命令例外ベクタを通して開始されます。このソフトウェアの開始は、*バウンス*と呼ばれます。

VFPv3U が実装されている場合は、バウンス機構を使用して、トラップされた浮動小数点例外がサポートされます。*トラップ*と呼ばれるトラップされた浮動小数点例外は、実装がアプリケーションソフトウェアに戻して解決を行う浮動小数点例外です。*浮動小数点例外* P. A2-42 を参照して下さい。サポートコードは、トラップされた例外をキャッチして、トラップハンドラ呼び出しに変換する必要があります。

サポートコードでは、実装で決定されたその他のタスクを実行することができます。このタスクは、ハードウェア実装が困難な動作や、ハードウェア内のゲートを大量に使用する動作などの稀な条件で使用される場合があります。これによって、さまざまなレベルのハードウェアサポートに対して一貫したソフトウェア動作が可能になります。

実装におけるハードウェアコンポーネントとソフトウェアコンポーネントとの役割分担、およびサポートコードとハードウェアとの間のインタフェースの詳細は、サブアーキテクチャ定義です。

非同期バウンス、直列化、VFP 例外バリア

VFP 実装では、以前の VFP 命令のサポートコード処理が必要なため、VFP 命令で未定義命令例外が取得される*非同期バウンス*を生成する可能性があります。必要な処理の性質がサポートコードに通知される機構は、サブアーキテクチャ定義です。通常は、次のような処理が含まれます。

- FPEXC のサブアーキテクチャ定義ビットを使用する。*浮動小数点例外レジスタ (FPEXC)* P. B1-68 を参照して下さい。
- サブアーキテクチャ定義の拡張機能システムレジスタを使用する。*アドバンスト SIMD および VFP 拡張機能のシステムレジスタ* P. B1-66 を参照して下さい。
- コンテキストスイッチでサブアーキテクチャ定義の拡張機能システムレジスタを保存する必要があることを示すため、FPEXC.EX == 1 にセットする。

非同期バウンスは、未定義命令例外の取得前に実行された最後の VFP 命令と無関係の場合があります。別の VFP 命令が発行され、非同期バウンスの発生前に終了した可能性もあります。この現象は、この途中の命令が、サポートコード処理を必要とする VFP 命令に対してレジスタ依存でない場合にのみ発生します。さらに、発行されて終了した途中の VFP 命令を処理するためのサブアーキテクチャ定義の機構が存在する可能性もあります。

ただし、FPSID、FPSCR、または FPEXC レジスタにアクセスする `VMRS` 命令と `VMSR` 命令は、*直列化*命令です。これは、これらの命令によって、レジスタ転送が実行される前に、サポートコード処理が必要な先行するすべての VFP 命令に含まれているすべての例外条件が検出され、拡張機能システムレジスタに反映されることが保証されることを意味します。読み出し専用 FPSID レジスタに対する `VMSR` 命令は、直列化 NOP です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In addition:

- A VMRS or VMSR instruction that accesses the FPSCR acts as a *VFP exception barrier*. This means it ensures that any outstanding exceptional conditions in preceding VFP instructions have been detected and processed by the support code before it performs the register transfer. If necessary, the VMRS or VMSR instruction takes an asynchronous bounce to force the processing of outstanding exceptional conditions.
- VMRS and VMSR instructions that access the FPSID or FPEXC do not take asynchronous bounces.

VFP serialization and the VFP exception barriers are described in pseudocode by the SerializeVFP() and VFPExcBarrier() functions respectively:

```
SerializeVFP()

VFPExcBarrier()
```

Interactions with the ARM architecture

ARM recommends that a VFP extension uses the Undefined Instruction mechanism to invoke its support code, see *Undefined Instruction exceptions* on page B1-76. To do this:

1. Before enabling the extension hardware, install the support code on the Undefined Instruction vector.
2. If the extension hardware requires assistance from the support code, it does not respond to a VFP instruction.
3. This causes an Undefined Instruction exception, that causes the support code to be executed.

VFP load/store instructions can generate Data Abort exceptions, and therefore implementations must be able to cope with a Data Abort exception on any memory access caused by such instructions.

Interrupts

Taking the Undefined Instruction exception causes IRQs to be disabled, see *Undefined Instruction exception* on page B1-49. Normally, IRQs are not re-enabled until the exception handler returns. This means that normal use of a VFP extension that requires support code in a system can increases worst case IRQ latency considerably.

You can reduce this IRQ latency penalty considerably by explicitly re-enabling interrupts soon after entry to the Undefined Instruction handler. This requires careful integration of the Undefined Instruction handler into the rest of the operating system. How this might be done is highly system-specific and beyond the scope of this manual.

A system where the IRQ handler itself might use the VFP coprocessor has a second potential cause of increased IRQ latency. This increase occurs if a long latency VFP operation is initiated by the interrupted application program, denying the use of the extension hardware to the IRQ handler for a significant number of cycles.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

さらに、次の規則が適用されます。

- FPSCR にアクセスする VMRS 命令または VMSR 命令は、*VFP 例外バリア*として機能します。これは、先行する VFP 命令内のすべての未解決の例外条件が、レジスタ転送が実行される前にサポートコードによって検出され、処理されることが保証されることを意味します。必要なら、VMRS 命令または VMSR 命令で非同期バウンスを取得して、未解決の例外条件を強制的に処理します。
- FPSID または FPEXC にアクセスする VMRS 命令と VMSR 命令では、非同期バウンスが取得されません。

VFP の直列化と VFP 例外バリアについては、それぞれ SerializeVFP() 関数と VFPExcBarrier() 関数の擬似コードで説明します。

```
SerializeVFP()

VFPExcBarrier()
```

ARM アーキテクチャとの相互動作

VFP 拡張機能では、未定義命令機構を使用してサポートコードを起動することを推奨します。*未定義命令例外* P. B1-76 を参照して下さい。これを実現するには、次の手順を実行します。

1. 拡張機能ハードウェアを許可する前に、未定義命令ベクタ上にサポートコードをインストールします。
2. 拡張機能ハードウェアでサポートコードからの支援が必要な場合、ハードウェアは VFP 命令に応答しません。
3. これによって、未定義命令例外が発生し、サポートコードが実行されます。

VFP ロード / ストア命令によってデータアボート例外が生成される可能性があるため、実装では、このような命令によって引き起こされたすべてのメモリアクセスでのデータアボート例外に対処できる必要があります。

割り込み

未定義命令例外が取得されると、IRQ は不可能になります。*未定義命令例外* P. B1-49 を参照して下さい。通常は、例外ハンドラが復帰するまで、IRQ は再び可能になりません。これは、システムのサポートコードを必要とする VFP 拡張機能の通常の使用で、ワーストケースの IRQ レイテンシが大幅に増加する可能性があることを意味します。

未定義命令ハンドラの開始直後に明示的に割り込みを再び可能にすることによって、この IRQ レイテンシペナルティを大幅に軽減することができます。このためには、未定義命令ハンドラをオペレーティングシステムの他の部分と注意深く統合する必要があります。この実現方法はシステム依存度が高く、本書の範囲を超えています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Therefore, if a system contains IRQ handlers that require both low interrupt latency and the use of VFP instructions, ARM recommends that the use of the highest latency Advanced SIMD or VFP instructions is avoided.

Note

FIQs are not disabled by entry to the Undefined Instruction handler, and so FIQ latency is not affected by the use of the Undefined Instruction exception described here. However, because they are not disabled, an FIQ can occur at any point during support code execution, including during the entry and exit sequences of the Undefined Instruction handler. If an FIQ handler can make any change to the state of the Advanced SIMD or VFP implementation, you must take great care to ensure that it handles every case correctly. Usually, this requirement is incompatible with the requirement that FIQs provide fast interrupt processing. Therefore ARM recommends that FIQ handlers do not use the Advanced SIMD or VFP extension.

B1.8.6 VFP subarchitecture support

In the ARMv7 specification of the VFP extension, some features are identified as SUBARCHITECTURE DEFINED. ARMv7 is fully compatible with the *ARM Common VFP subarchitecture*, that ARM has used for several VFP implementations. However, ARMv7 does not require or specifically recommend the use of the ARM Common VFP subarchitecture.

Appendix B *Common VFP Subarchitecture Specification* is the specification of the *ARM Common VFP subarchitecture*. The subarchitecture is not part of the ARMv7 architecture specification. For details of the status of the subarchitecture specification see the *Note* on the cover page of Appendix B.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

IRQ ハンドラ自体で VFP コプロセッサが使用されるシステムには、IRQ レイテンシを増加させる第 2 の潜在的な原因があります。割り込まれたアプリケーションプログラムによって長いレイテンシの VFP 動作が開始され、IRQ ハンドラが多くのサイクルの間、拡張機能ハードウェアの使用を拒否された場合、IRQ レイテンシが増加します。

したがって、システムに低割り込みレイテンシと VFP 命令の使用の両方が必要な IRQ ハンドラが含まれている場合は、レイテンシが最も長いアドバンスト SIMD 命令または VFP 命令は使用しないことをお勧めします。

注

未定義命令ハンドラが開始されても FIQ は禁止されないため、FIQ レイテンシは、ここで説明されている未定義命令例外の使用による影響を受けません。ただし、FIQ が禁止されていないため、未定義命令ハンドラの開始シーケンスと終了シーケンスを含む、サポートコード実行期間の任意の時点で FIQ が発生する可能性があります。FIQ ハンドラでアドバンスト SIMD 実装または VFP 実装の状態を変更できる場合は、すべてのケースが正しく処理されることを保証するため、十分な注意が必要です。多くの場合、この要件は、FIQ が高速割り込み処理を提供する要件に適合しません。そのため、FIQ ハンドラではアドバンスト SIMD 拡張機能または VFP 拡張機能を使用しないことをお勧めします。

B1.8.6 VFP サブアーキテクチャのサポート

VFP 拡張機能の ARMv7 仕様では、一部の機能がサブアーキテクチャ定義とされています。ARMv7 は、いくつかの VFP 実装で使用されている *ARM 共通 VFP サブアーキテクチャ*と完全に互換性があります。ただし、ARMv7 で、ARM 共通 VFP サブアーキテクチャの使用が要求されたり、特別に推奨されたりすることはありません。

*ARM 共通 VFP サブアーキテクチャ*の仕様を、付録 B *「一般的な VFP サブアーキテクチャ仕様」*に示します。このサブアーキテクチャは、ARMv7 アーキテクチャ仕様の一部ではありません。このサブアーキテクチャ仕様のステータスの詳細については、付録 B の表紙の*注*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.9 Execution environment support

Support code for an execution environment can execute in two of the processor states described in *Instruction set states* on page B1-23:

- ThumbEE state supports the Thumb Execution Environment. For more information, see *Thumb Execution Environment*.
- Jazelle state supports direct bytecode execution. For more information, see *Jazelle direct bytecode execution* on page B1-74.

B1.9.1 Thumb Execution Environment

See *Thumb Execution Environment* on page A2-69 for an introduction to the *Thumb Execution Environment* (ThumbEE), including an application level view of the execution environment, and a definition of its CP14 registers. This section describes the system level programmers' model for ThumbEE. For more information about ThumbEE see Chapter A9 *ThumbEE*.

The ThumbEE Configuration Register can be read in User mode, but can be written only in privileged modes, see *ThumbEE Configuration Register (TEECR)* on page A2-70.

Access to the ThumbEE Handler Base Register depends on the value held in the TEECR and the current privilege level, see *ThumbEE Handler Base Register (TEEHBR)* on page A2-71.

The processor executes ThumbEE instructions when it is in ThumbEE state.

The processor instruction set state is indicated by the CPSR.J and CPSR.T bits, see *Program Status Registers (PSRs)* on page B1-14. (J,T) == 0b11 when the processor is in ThumbEE state.

During normal execution, not involving exception entries and returns:

- ThumbEE state can only be entered from Thumb state, using the `ENTERX` instruction
- exit from ThumbEE state always occurs using the `LEAVEX` instruction and returns execution to Thumb state.

For details of these instructions see *ENTERX, LEAVEX* on page A9-7.

When an exception occurs in ThumbEE state, exception entry goes to either ARM state or Thumb state as usual, depending on the value of SCTLR.TE. When the exception handler returns, the exception return instruction restores CPSR.J and CPSR.T as usual, causing a return to ThumbEE state.

In ThumbEE state, execution of the exception return instructions described in *Exception return* on page B1-38 is UNPREDICTABLE.

ThumbEE and the Security Extensions

When an implementation that supports ThumbEE includes the Security Extensions, the ThumbEE registers are not banked. If ThumbEE support is required in both Secure and Non-secure states, the monitor must save and restore the register contents accordingly.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B1.9 実行環境のサポート

実行環境のサポートコードは、*命令セット状態* P. B1-23 で説明されている 2 つのプロセッサ状態で実行することができます。

- ThumbEE 状態では、Thumb 実行環境がサポートされます。詳細については、*Thumb 実行環境*を参照して下さい。
- Jazelle 状態では、直接バイトコード実行がサポートされます。詳細については、*Jazelle 直接バイトコード実行* P. B1-74 を参照して下さい。

B1.9.1 Thumb 実行環境

実行環境のアプリケーションレベルでのビューと CP14 レジスタの定義を含む *Thumb 実行環境* (ThumbEE) の概要については、*Thumb 実行環境* P. A2-69 を参照して下さい。このセクションでは、ThumbEE のシステムレベルのプログラマモデルについて説明します。ThumbEE の詳細については、第 A9 章*「ThumbEE」*を参照して下さい。

ThumbEE 構成レジスタは、ユーザモードで読み出すことができますが、特権モードでしか書き込むことができません。*ThumbEE 構成レジスタ (TEECR)* P. A2-70 を参照して下さい。

ThumbEE ハンドラのベースレジスタへのアクセスは、TEECR に保存された値と現在の特権レベルに依存します。*ThumbEE ハンドラ ベースレジスタ (TEEHBR)* P. A2-71 を参照して下さい。

プロセッサは、ThumbEE 状態にあるときに ThumbEE 命令を実行します。

プロセッサの命令セット状態は、CPSR.J ビットと CPSR.T ビットによって示されます。*プログラムステータス レジスタ (PSR)* P. B1-14 を参照して下さい。プロセッサが ThumbEE 状態にあるときは、(J,T) == 0b11 です。

例外の開始と復帰を伴わない通常の実行中、次の規則が適用されます。

- ThumbEE 状態は、`ENTERX` 命令を使用して、Thumb 状態からのみ開始することができます。
- ThumbEE 状態の終了は、必ず `LEAVEX` 命令を使用して行われ、実行が Thumb 状態に戻ります。

これらの命令の詳細については、*ENTERX*、*LEAVEX* P. A9-7 を参照して下さい。

ThumbEE 状態で例外が発生すると、例外の開始が、SCTLR.TE の値に応じて、ARM 状態と Thumb 状態のどちらかに遷移します。例外ハンドラが復帰するときに、例外からの復帰命令によって通常に CPSR.J と CPSR.T が復元され、ThumbEE 状態に戻ります。

ThumbEE 状態では、*例外からの復帰* P. B1-38 で説明されている例外からの復帰命令の実行が予測不能になります。

ThumbEE とセキュリティ拡張機能

ThumbEE をサポートする実装にセキュリティ拡張機能が含まれている場合は、ThumbEE レジスタがバンクされません。ThumbEE サポートがセキュア状態と非セキュア状態の両方で必要な場合は、モニタでレジスタの内容を保存して復元する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Aborts, exceptions, and checks

Aborts and exceptions are unchanged in ThumbEE. A null check takes priority over an abort or watchpoint on the same memory access. For more information, see *Null checking* on page A9-3.

The IT state bits in the CPSR are always cleared on entry to a NullCheck or IndexCheck handler. For more information, see *IT block and check handlers* on page A9-5.

B1.9.2 Jazelle direct bytecode execution

In Jazelle state the processor executes bytecode programs, as described in *Jazelle state* on page A2-74.

The processor instruction set state is indicated by the CPSR.J and CPSR.T bits, see *Program Status Registers (PSRs)* on page B1-14. (J,T) == 0b10 when the processor is in Jazelle state. For more information about entering and leaving Jazelle state see *Jazelle state* on page B1-81.

Extension of the PC to 32 bits

To enable the PC to point to an arbitrary bytecode instruction, in a non-trivial Jazelle implementation all 32 bits of the PC are defined. In the PC, bit [0] always reads as zero when in ARM, Thumb, or ThumbEE state.

The existence of bit [0] in the PC is only visible in ARM, Thumb, or ThumbEE states when an exception occurs in Jazelle state and the exception return address is odd-byte aligned.

The main architectural implication of this is that an exception handler must ensure that it restores all 32 bits of the PC. The recommended ways of handling exception returns behave correctly.

Exception handling in the Jazelle extension

Exceptions on page B1-30 describes how exception entry occurs if an exception occurs while the processor is executing in Jazelle state. This section gives more information about how exceptions in Jazelle state are taken and handled.

Interrupts and Fast interrupts, IRQ and FIQ

To enable the standard mechanism for handling interrupts to work correctly, a Jazelle hardware implementation must ensure that one of the following applies at the point where execution of a bytecode instruction might be interrupted by an IRQ or FIQ:

- Execution has reached a bytecode instruction boundary. That is:
  - — all operations required to implement one bytecode instruction have completed
  - — no operations required to implement the next bytecode instruction has completed.

  The LR value on entry to the interrupt handler must be (address of the next bytecode instruction) + 4.
- The sequence of operations performed from the start of execution of the current bytecode instruction, up to the point where the interrupt occurs, is idempotent. This means that the sequence can be repeated from its start without changing the overall result of executing the bytecode instruction.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アボート、例外、チェック

アボートと例外は、ThumbEE 状態では変化しません。NULL チェックは、同じメモリアクセスでのアボートやウォッチポイントよりも優先されます。詳細については、*NULL チェック* P. A9-3 を参照して下さい。

CPSR 内の IT 状態ビットは、NullCheck ハンドラまたは IndexCheck ハンドラの開始時点に常にクリアされます。詳細については、*IT ブロックおよびチェックハンドラ* P. A9-5 を参照して下さい。

B1.9.2 Jazelle 直接バイトコード実行

Jazelle 状態 P. A2-74 で説明されているように、Jazelle 状態ではプロセッサはバイトコードのプログラムを実行します。

プロセッサの命令セット状態は、CPSR.J ビットと CPSR.T ビットによって示されます。*プログラムステータス レジスタ (PSR)* P. B1-14 を参照して下さい。プロセッサが Jazelle 状態にあるときは、(J,T) = 0b10 です。Jazelle 状態の開始と終了の詳細については、*Jazelle 状態* P. B1-81 を参照して下さい。

PC の 32 ビットへの拡張

PC ですべてのバイトコード命令を指すことができるように、非トリビアル Jazelle 実装では、PC の 32 ビットすべてが定義されます。ARM 状態、Thumb 状態、ThumbEE 状態では、PC のビット [0] は常に 0 として読み出されます。

Jazelle 状態で例外が発生し、例外からの復帰アドレスが奇数バイトにアラインされている場合は、ARM 状態、Thumb 状態、または ThumbEE 状態でしか PC のビット [0] の存在を認識できません。

この重要なアーキテクチャ上の意味は、例外ハンドラで PC の 32 ビットすべてが復元されることを保証する必要があるということです。推奨の例外処理方法では、復帰が正しく動作します。

Jazelle 拡張機能での例外処理

例外 P. B1-30 では、プロセッサが Jazelle 状態で実行中に例外が発生した場合、どのように例外が開始されるかについて説明します。このセクションでは、Jazelle 状態での例外の取得と処理の方法について詳しく説明します。

割り込みと高速割り込み（IRQ と FIQ）

標準的な割り込み処理機構を正しく機能させるためには、Jazelle ハードウェア実装において、バイトコード命令の実行が IRQ または FIQ によって中断された時点で、次のいずれかが適用されることを保証する必要があります。

- 実行がバイトコード命令の境界に到達している。これは、次のことを意味します。
  - – 1 つのバイトコード命令を実装するために必要なすべての動作が完了している。
  - – 次のバイトコード命令を実装するために必要な、どの動作も完了していない。

  割り込みハンドラ開始時の LR の値は、（次のバイトコード命令のアドレス + 4）であることが必要です。
- 現在のバイトコード命令の実行の開始から、割り込みが発生する時点までに実行される動作シーケンスがベキ等である。これは、そのシーケンスが、バイトコード命令の全体的な実行結果を変更することなく、最初から繰り返し可能であることを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The LR value on entry to the interrupt handler must be (address of the current bytecode instruction) + 4.

- Corrective action is taken either:
  - — directly by the Jazelle extension hardware
  - — indirectly, by calling a SUBARCHITECTURE DEFINED handler in the EJVM.

  The corrective action must re-create a situation where the bytecode instruction can be re-executed from its start.

  The LR value on entry to the interrupt handler must be (address of the interrupted bytecode instruction) + 4.

Data Abort exceptions

On taking a Data Abort exception, the value saved in LR_abt must ensure that the Data Abort handler can:

- read the CP15 Fault Status and Fault Address registers
- fix the reason for the abort
- return using `SUBS PC,LR,#8` or its equivalent.

The abort handler must be able to do this without looking at the instruction that caused the abort or which instruction set state it was executed in. On an ARMv7-A implementation, the abort handler must take account of the virtual memory system.

Note

- This assumes that the intention is to return to and retry the bytecode instruction that caused the Data Abort exception. If the intention is instead to return to the bytecode instruction after the one that caused the abort, then the return address must be modified by the length of the bytecode instruction that caused the abort.
- For details of the CP15 Fault Status and Fault Address:
  - — for a VMSA implementation, see *CP15 c5, Fault status registers* on page B3-121 and *CP15 c6, Fault Address registers* on page B3-124
  - — for a PMSA implementation, see *CP15 c5, Fault status registers* on page B4-54 and *CP15 c6, Fault Address registers* on page B4-57.

To enable the standard mechanism for handling Data Abort exceptions to work correctly, a Jazelle hardware implementation must ensure that one of the following applies at any point where a bytecode instruction can generate a Data Abort exception:

- The sequence of operations performed from the start of execution of the bytecode instruction, up to the point where the Data Abort exception is generated, is idempotent. This means that the sequence can be repeated from its start without changing the overall result of executing the bytecode instruction.
- If the Data Abort exception is generated during execution of a bytecode instruction, corrective action is taken either:
  - — directly by the Jazelle extension hardware

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

割り込みハンドラ開始時の LR の値は、（現在のバイトコード命令のアドレス + 4）であることが必要です。

- 次のいずれかの方法で修正動作が行われる。
  - – Jazelle 拡張機能ハードウェアで直接的に
  - – EJVM 内でサブアーキテクチャ定義のハンドラを呼び出すことによって間接的に

  修正動作では、バイトコード命令が最初から再実行できるような状況を再作成する必要があります。

  割り込みハンドラの開始時点の LR の値は、（割り込まれたバイトコード命令のアドレス + 4）であることが必要です。

データアボート例外

データアボート例外を取得した時点で、LR_abt に保存されている値によって、データアボートハンドラが次の動作を実行可能なことを保証する必要があります。

- CP15 のフォールトステータスおよびフォールトアドレス レジスタの読み出し
- アボートの原因の修正
- `SUBS PC,LR,#8` または同等のコードを使用した復帰

アボートハンドラは、アボートを引き起こした命令、またはそのときの命令セット状態を確認せずにこれらの動作を実行できる必要があります。ARMv7-A 実装では、アボートハンドラは仮想メモリシステムを考慮する必要があります。

注

- これは、データアボート例外を引き起こしたバイトコード命令に復帰して再試行することが目的であるという前提です。そうではなく、アボートを引き起こしたバイトコード命令よりも後の命令に復帰することが目的の場合は、アボートを引き起こしたバイトコード命令の長さだけ復帰アドレスを修正する必要があります。
- CP15 のフォールトステータスとフォールトアドレスの詳細については、次のセクションを参照して下さい。
  - – *CP15 c5、フォールトステータス レジスタ* P. B3-121 と *CP15 c6、フォールトアドレス レジスタ* P. B3-124（VMSA 実装の場合）
  - – *CP15 c5、フォールトステータス レジスタ* P. B4-54 と *CP15 c6、フォールトアドレス レジスタ* P. B4-57（PMSA 実装の場合）

標準的なデータアボート例外処理機構を正しく機能させるためには、Jazelle ハードウェア実装において、バイトコード命令によってデータアボート例外が生成される可能性があるすべての時点で、次のいずれかが適用されることを保証する必要があります。

- バイトコード命令の実行の開始から、データアボート例外が生成される時点までに実行される動作シーケンスがベキ等である。これは、そのシーケンスが、バイトコード命令の全体的な実行結果を変更することなく、最初から繰り返し可能であることを意味します。
- データアボート例外がバイトコード命令の実行中に生成された場合は、次のいずれかの方法で修正動作が行われる。
  - – Jazelle 拡張機能ハードウェアで直接的に

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

— indirectly, by calling a SUBARCHITECTURE DEFINED handler in the EJVM.

The corrective action must re-create a situation where the bytecode instruction can be re-executed from its start.

Note

From ARMv6, the ARM architecture does not support the Base Updated Abort Model. This removes a potential obstacle to the first of these solutions. For information about the Base Updated Abort Model in earlier versions of the ARM architecture see *The ARM abort model* on page AppxH-20.

Prefetch Abort exceptions

On taking a Prefetch Abort exception, the value saved in LR_abt must ensure that the Prefetch Abort handler can locate the start of the instruction that caused the abort simply and without looking at the instruction set state in which its execution was attempted. The start of this instruction is always at address (LR_abt – 4). On an ARMv7-A implementation, the abort handler must take account of the virtual memory system.

A multi-byte bytecode instruction can cross a page boundary. In this case the Prefetch Abort handler cannot use LR_abt to determine which of the two pages caused the abort. How this situation is handled is SUBARCHITECTURE DEFINED, but if it is handled by taking a Prefetch Abort exception, the architecture requires that (LR_abt – 4) must point to the first byte of the bytecode instruction that caused the abort.

To ensure subarchitecture-independence, OS designers must write Prefetch Abort handlers in such a way that they can handle a Prefetch Abort exception generated in either of the two pages spanned by a multi-byte bytecode instruction that crosses a page boundary. In an implementation that has an Instruction Fault Address Register (IFAR), the IFAR can be used to determine the faulting page. Otherwise, a simple technique is:

```
IF the page pointed to by (LR_abt - 4) is not mapped
    THEN map the page
    ELSE map the page following the page including (LR_abt - 4)
ENDIF
retry the instruction
```

SVC and SMC exceptions

SVC and SMC exceptions must not be taken during Jazelle state execution. To cause either of these exceptions to be taken, a Jazelle implementation must exit to a software handler that executes an `SVC` or `SMC` instruction.

Undefined Instruction exceptions

The Undefined Instruction exception must not be taken during Jazelle state execution, except on a trivial implementation of Jazelle state as described in *Exception return to an unsupported instruction set state* on page B1-40.

When executing in Jazelle state, the Jazelle extension hardware might use a coprocessor extension such as the VFP extension to execute some operations. If it does so, it must avoid taking Undefined Instruction exceptions while in Jazelle state, even if an exceptional condition occurs that would normally cause the coprocessor extension to generate an Undefined Instruction exception.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

– EJVM 内でサブアーキテクチャ定義のハンドラを呼び出すことによって間接的に

修正動作では、バイトコード命令が最初から再実行できるような状況を再作成する必要があります。

注

ARMv6 およびそれ以降の ARM アーキテクチャでは、ベース更新アボートモデルがサポートされていません。これによって、初期のこれらのソリューションの潜在的な障害が取り除かれます。以前のバージョンの ARM アーキテクチャにおけるベース更新アボートモデルの詳細については、*ARM アボートモデル* P. 付録 H-20 を参照して下さい。

プリフェッチアボート例外

プリフェッチアボート例外の取得時点で、LR_abt に保存されている値によって、プリフェッチアボート ハンドラが、実行の試みられた命令セット状態を確認せず、アボートを引き起こした命令の先頭を簡単に特定できることを保証する必要があります。この命令の先頭は、常にアドレス (LR_abt – 4) です。ARMv7-A 実装では、アボートハンドラが仮想メモリシステムを考慮する必要があります。

マルチバイトのバイトコード命令は、ページ境界をまたぐ可能性があります。この場合、プリフェッチアボート ハンドラは、LR_abt を使用して、どちらのページでアボートが引き起こされたかを特定できません。この状況の処理方法はサブアーキテクチャ定義ですが、プリフェッチアボート例外を取得して処理する場合は、アーキテクチャ上、(LR_abt – 4) がアボートを引き起こしたバイトコード命令の最初のバイトを指している必要があります。

サブアーキテクチャに依存しないことを保証するために、OS 設計者は、ページ境界をまたぐマルチバイトのバイトコード命令について、2 つのページのどちらで生成されたプリフェッチアボート例外でも処理できるように、プリフェッチアボート ハンドラを作成する必要があります。命令フォールトアドレス レジスタ (IFAR) が存在する実装では、IFAR を使用して障害ページを特定することができます。それ以外の場合は、次のような単純なテクニックが使用されます。

```
IF the page pointed to by (LR_abt - 4) is not mapped
    THEN map the page
    ELSE map the page following the page including (LR_abt - 4)
ENDIF
retry the instruction
```

SVC 例外と SMC 例外

SVC 例外と SMC 例外は、Jazelle 状態実行中に取得できません。これらの例外のどちらかを取得するには、Jazelle 実装で、`SVC` 命令または `SMC` 命令を実行するソフトウェアハンドラに遷移する必要があります。

未定義命令例外

未定義命令例外は、*サポートされていない命令セット状態への例外からの復帰* P. B1-40 で説明されているような Jazelle 状態のトリビアル実装の場合を除いて、Jazelle 状態実行中に取得できません。

Jazelle 状態での実行中は、Jazelle 拡張機能ハードウェアが、VFP 拡張機能などのコプロセッサ拡張機能を使用して何らかの動作を実行している可能性があります。その場合は、通常ではコプロセッサ拡張機能によって未定義命令例外が生成されると思われる例外状態が発生しても、Jazelle 状態での未定義命令例外の取得を避ける必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Jazelle state configuration and control

For details of the configuration and control of Jazelle state from the application level, see *Application level configuration and control of the Jazelle extension* on page A2-75. That section includes descriptions of the Jazelle extension registers that can be accessed from User mode:

- *Jazelle ID Register (JIDR)* on page A2-76
- *Jazelle Main Configuration Register (JMCR)* on page A2-77.

The other Jazelle extension register is accessible only from privileged modes, see *Jazelle OS Control Register (JOSCR)*. This register controls access to the Jazelle extension.

When the Security Extensions are implemented, the Jazelle registers are common to the Secure and Non-secure security states. Each register has the same access permissions in both security states. For more information, see the register descriptions.

Changes to the Jazelle CP14 registers have the same synchronization requirements as changes to the CP15 registers. These are described in:

- *Changes to CP15 registers and the memory order model* on page B3-77 for a VMSA implementation
- *Changes to CP15 registers and the memory order model* on page B4-28 for a PMSA implementation.

Note

- Normally, an EJVM never accesses the JOSCR.
- An EJVM that runs in User mode must not attempt to access the JOSCR.

Jazelle OS Control Register (JOSCR)

The Jazelle OS Control Register (JOSCR) provides operating system control of the use of the Jazelle extension by processes and threads.

The JOSCR is:

- a CP14 register
- a 32-bit read/write register
- accessible only from privileged modes
- when the Security Extensions are implemented, a Common register.

The format of the JOSCR is:

31 … 2	1	0
Reserved, RAZ	CV	CD

Bits [31:2] Reserved, RAZ. These bits are reserved for future expansion.

CV, bit [1] Configuration Valid bit. This bit is used by an operating system to signal to the EJVM that it must re-write its configuration to the configuration registers. The possible values are:

0 Configuration not valid. The EJVM must re-write its configuration to the configuration registers before it executes another bytecode instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Jazelle 状態の構成と制御

アプリケーションレベルからの Jazelle 状態の構成と制御の詳細については、*アプリケーションレベルの構成と Jazelle 拡張機能の制御* P. A2-75 を参照して下さい。このセクションには、ユーザモードからアクセス可能な Jazelle 拡張機能レジスタの説明が記載されています。

- *JazelleID レジスタ (JIDR)* P. A2-76
- *Jazelle メイン構成レジスタ (JMCR)* P. A2-77

その他の Jazelle 拡張機能レジスタには、特権モードでのみアクセスできます。*Jazelle OS 制御レジスタ (JOSCR)* を参照して下さい。このレジスタによって、Jazelle 拡張機能へのアクセスが制御されます。

セキュリティ拡張機能が実装されている場合は、Jazelle レジスタがセキュアと非セキュアのセキュリティ状態で共通です。それぞれのレジスタには、両方のセキュリティ状態で同じアクセス許可が与えられます。詳細については、レジスタの説明を参照して下さい。

Jazelle CP14 レジスタの変更には、CP15 レジスタの変更と同様の同期要件が適用されます。これらについては、次のセクションを参照して下さい。

- *CP15 レジスタの変更とメモリオーダ モデル* P. B3-77（VMSA 実装の場合）
- *CP15 レジスタの変更とメモリオーダ モデル* P. B4-28 （PMSA 実装の場合）

注

- 通常は、EJVM から JOSCR にアクセスすることはありません。
- ユーザモードで実行中の EJVM から JOSCR にアクセスを試みないで下さい。

Jazelle OS 制御レジスタ (JOSCR)

Jazelle OS 制御レジスタ (JOSCR) では、プロセッサとスレッドによる Jazelle 拡張機能の使用のオペレーティングシステム制御が提供されます。

JOSCR の特徴は次のとおりです。

- CP14 レジスタ
- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードからのみアクセス可能
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

JOSCR のフォーマットは次のとおりです。

31 … 2	1	0
予約、RAZ	CV	CD

ビット [31:2] 予約、RAZ。これらのビットは将来の拡張のために予約済みです。

CV、ビット [1] 構成有効ビット。このビットは、オペレーティングシステムにより、その構成を構成レジスタに再書き込みする必要があることを EJVM に通知するために使用されます。可能な値は次のとおりです。

0 構成が有効ではありません。EJVM は、他のバイトコード命令を実行する前に、その構成を構成レジスタに再書み込みする必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

1 Configuration valid. The EJVM does not need to update the configuration registers.

When the JMCR.JE bit is set to 1, the CV bit also controls entry to Jazelle state, see *Controlling entry to Jazelle state* on page B1-79.

CD, bit [0] Configuration Disabled bit. This bit is used by an operating system to disable User mode access to the JIDR and configuration registers:

0 Configuration enabled. Access to the Jazelle registers, including User mode accesses, operate normally. For more information, see the register descriptions in *Application level configuration and control of the Jazelle extension* on page A2-75.

1 Configuration disabled in User mode. User mode access to the Jazelle registers are UNDEFINED, and all User mode accesses to the Jazelle registers cause an Undefined Instruction exception.

For more information about the use of this bit see *Monitoring and controlling User mode access to the Jazelle extension* on page B1-80.

The JOSCR provides a control mechanism that is independent of the subarchitecture of the Jazelle extension. An operating system can use this mechanism to control access to the Jazelle extension. Normally, this register is used in conjunction with the JMCR.JE bit, see *Jazelle Main Configuration Register (JMCR)* on page A2-77.

The JOSCR.CV and JOSCR.CD bits are both set to 0 on reset. This ensures that, subject to some conditions, an EJVM can operate under an OS that does not support the Jazelle extension. The main condition required to ensure an EJVM can operate under an OS that does not support the Jazelle extension it that the operating system never swaps between two EJVM processes that require different settings of the Jazelle configuration registers.

Two examples of how this condition can be met in a system are:

- if there is only ever one process or thread using the EJVM
- if all of the processes or threads that use the EJVM use the same static settings of the configuration registers.

Accessing the JOSCR

To access the JOSCR you read or write the CP14 registers with `<opc1>` set to 7, `<CRn>` set to c1, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC  p14, 7, <Rt>, c1, c0, 0   ; Read  Jazelle OS Control Register
MCR  p14, 7, <Rt>, c1, c0, 0   ; Write Jazelle OS Control Register
```

Note

For maximum compatibility with any future enhancements to the Jazelle extension, ARM strongly recommends that a read, modify, write sequence is used to update the JOSCR. Updating the register in this way preserves the value of any of bits [31:2] that might be used by a future expansion.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

1 構成が有効です。EJVM は、構成レジスタを更新する必要がありません。

JMCR.JE ビットが 1 に設定されている場合は、CV ビットによっても Jazelle 状態の開始が制御されます。*Jazelle 状態の開始の制御* P. B1-79 を参照して下さい。

CD、ビット [0] 構成不可能ビット。このビットは、オペレーティングシステムで、JIDR と構成レジスタへのユーザモード アクセスを不可能にするために使用されます。

0 構成が可能です。ユーザモード アクセスを含む Jazelle レジスタへのアクセスが正常に動作します。詳細については、*アプリケーションレベルの構成と Jazelle 拡張機能の制御* P. A2-75 のレジスタの説明を参照して下さい。

1 ユーザモードで構成が不可能になっています。Jazelle レジスタへのユーザモード アクセスは未定義で、Jazelle レジスタへのユーザモード アクセスはすべて未定義命令例外を引き起します。

このビットの使用方法については、*Jazelle 拡張機能へのユーザモード アクセスの監視と制御* P. B1-80 を参照して下さい。

JOSCR では、Jazelle 拡張機能のサブアーキテクチャとは無関係の制御機構が提供されます。オペレーティングシステムは、この機構を使用して Jazelle 拡張機能へのアクセスを制御することができます。通常、このレジスタは、JMCR.JE ビットと組み合わせて使用されます。*Jazelle メイン構成レジスタ (JMCR)* P. A2-77 を参照して下さい。

JOSCR.CV ビットと JOSCR.CD ビットはどちらもリセット時に 0 にセットされます。これによって、特定の条件下で、EJVM は、Jazelle 拡張機能をサポートしていない OS の下で動作することができます。EJVM が Jazelle 拡張機能をサポートしていない OS の下で動作できることを保証するために必要な主な条件は、オペレーティングシステムで、Jazelle 構成レジスタの異なる設定を必要とする 2 つの EJVM プロセス間で決してスワップを行わないことです。

この条件をシステムで満たす方法について、2 つの例を次に示します。

- EJVM を使用するプロセスまたはスレッドが常に 1 つしか存在しない場合
- EJVM を使用するすべてのプロセスまたはスレッドが、構成レジスタの同じ静的設定を使用する場合

JOSCR へのアクセス

JOSCR にアクセスするには、`<opc1>` を 7、`<CRn>` を c1、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP14 レジスタを読み出すか、書き込みます。次に例を示します。

```
MRC  p14, 7, <Rt>, c1, c0, 0   ; Read  Jazelle OS Control Register
MCR  p14, 7, <Rt>, c1, c0, 0   ; Write Jazelle OS Control Register
```

注

Jazelle 拡張機能に対する将来の機能強化との間で最大限の互換性を確保するため、JOSCR の更新には読み出し – 変更 – 書き込みシーケンスを使用することを強く推奨します。この方法でレジスタを更新すると、将来の機能強化で使用される可能性のあるビット [31:2] の値が保存されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Controlling entry to Jazelle state

The normal method of entering Jazelle state is using the BXJ instruction, see *Jazelle state entry instruction, BXJ* on page A2-74. The operation of this instruction depends on both:

- the value of the JMCR.JE bit, see *Jazelle Main Configuration Register (JMCR)* on page A2-77
- the value of the JOSCR.CV bit.

When the JMCR.JE bit is 0, the JOSCR has no effect on the execution of BXJ instructions. They always execute as BX instructions, and there is no attempt to enter Jazelle state.

When the JMCR.JE bit is 1, the JOSCR.CV bit controls the operation of BXJ instructions:

If CV == 1 The Jazelle extension hardware configuration is valid and enabled. A BXJ instruction causes the processor to enter Jazelle state in SUBARCHITECTURE DEFINED circumstances, and execute bytecode instructions as described in *Executing BXJ with Jazelle extension enabled* on page A2-75.

If CV == 0 The Jazelle extension hardware configuration is not valid and therefore entry to Jazelle state is disabled.

In all SUBARCHITECTURE DEFINED circumstances where, if CV had been 1 the BXJ instruction would have caused the Jazelle extension hardware to enter Jazelle state, it instead:

- enters a Configuration Invalid handler
- sets CV to 1.

A Configuration Invalid handler is a sequence of instructions that:

- includes MCR instructions to write the configuration required by the EJVM
- ends with a BXJ instruction to re-attempt execution of the required bytecode instruction.

The following are SUBARCHITECTURE DEFINED:

- how the address of the Configuration Invalid handler is determined
- the entry and exit conditions of the Configuration Invalid handler.

In circumstances in which the Jazelle extension hardware would not have entered Jazelle state if CV had been 1, it is IMPLEMENTATION DEFINED whether:

- the Configuration Invalid handler is entered
- a SUBARCHITECTURE DEFINED handler is entered, as described in *Executing BXJ with Jazelle extension enabled* on page A2-75.

In ARMv7, the JOSCVR.CV bit is set to 0 on exception entry for all implementations other than a trivial implementation of the Jazelle extension.

The intended use of the JOSCR.CV bit is:

1. When a context switch occurs, JOSCR.CV is set to 0. This is done by the operating system or, in ARMv7, as the result of an exception.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Jazelle 状態の開始の制御

Jazelle 状態を開始する通常の方法は、BXJ 命令の使用です。*Jazelle 状態開始命令、BXJ* P. A2-74 を参照して下さい。この命令の動作は次の両方に依存します。

- JMCR.JE ビットの値（*Jazelle メイン構成レジスタ (JMCR)* P. A2-77 を参照）
- JOSCR.CV ビットの値

JMCR.JE ビットが 0 の場合は、JOSCR が BXJ 命令の実行に影響を与えることはありません。この命令は常に BX 命令として実行され、Jazelle 状態の開始が試みられることはありません。

JMCR.JE ビットが 1 の場合は、JOSCR.CV ビットによって BXJ 命令の動作が制御されます。

CV == 1 の場合

Jazelle 拡張機能ハードウェアの構成が有効で、可能です。*Jazelle 拡張機能が稼働中である場合の BXJ の実行* P. A2-75 で説明されているように、BXJ 命令によって、プロセッサがサブアーキテクチャ定義の環境で Jazelle 状態を開始して、バイトコード命令を実行します。

CV == 0 の場合

Jazelle 拡張機能ハードウェアの構成が無効なため、Jazelle 状態の開始が不可能です。

CV が 1 であると仮定した場合に BXJ 命令によって Jazelle 拡張機能ハードウェアが Jazelle 状態を開始する、すべてのサブアーキテクチャ定義の状況では、代わりに次の動作が実行されます。

- 構成無効ハンドラを開始する。
- CV を 1 にセットする。

構成無効ハンドラは、次のような命令シーケンスですす。

- EJVM に必要な構成を書き込むための MCR 命令を含む。
- 必要なバイトコード命令の実行を再度試みるために BXJ 命令で終了する。

次のものはサブアーキテクチャ定義です。

- 構成無効ハンドラのアドレスの特定方法
- 構成無効ハンドラの開始条件と終了条件

CV が 1 であると仮定した場合も Jazelle 拡張機能ハードウェアが Jazelle 状態を開始しない状況では、次の動作が行われるかどうかは実装定義です。

- 構成無効ハンドラが開始される。
- *Jazelle 拡張機能が稼働中である場合の BXJ の実行* P. A2-75 で説明されているように、サブアーキテクチャ定義のハンドラが開始される。

ARMv7 では、Jazelle 拡張機能のトリビアル実装以外のすべての実装で、例外の開始時に JOSCVR.CV ビットが 0 にセットされます。

JOSCR.CV ビットの意図された使用法は次のとおりです。

1. コンテキストスイッチが発生すると、JOSCR.CV が 0 にセットされます。この処理はオペレーティングシステムによって、または ARMv7 では例外の結果として実行されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

2. When the new process or thread performs a BXJ instruction to start executing bytecode instructions, the Configuration Invalid handler is entered and JOSCR.CV becomes 1.

3. The Configuration Invalid handler:
   - writes the configuration required by the EJVM to the Jazelle configuration registers
   - retries the BXJ instruction to execute the bytecode instruction.

This ensures that the Jazelle extension configuration registers are set up correctly for the EJVM concerned before any bytecode instructions are executed. It successfully handles cases where a context switch occurs during execution of the Configuration Invalid handler.

Monitoring and controlling User mode access to the Jazelle extension

The system can use the JOSCR.CD bit in different ways to monitor and control User mode access to the Jazelle extension hardware. Possible uses include:

- An OS can set JOSCR.CD == 1 and JMCR.JE == 0, to prevent all User mode access to the Jazelle extension hardware. With these settings any use of the BXJ instruction has the same result as a BX instruction, and any attempt to configure the hardware, including any attempt to set the JMCR.JE bit to 1, results in an Undefined Instruction exception.

- A simple mechanism for the OS to provide User mode access to the Jazelle extension hardware, while protecting EJVMs from conflicting use of the hardware by other processes, is:
  - Set the JOSCR.CD bit to 0.
  - Preserve and restore the JMCR on context switches, initializing its value to 0 for new processes.
  - The JOSCR.CV bit is set to 0 on each context switch, either by the operating system or, in ARMv7, as the result of an exception. This ensures that EJVMs reconfigure the Jazelle extension hardware to match their requirements when necessary.

  The context switch mechanism is described in *Controlling entry to Jazelle state* on page B1-79.

EJVM operation

EJVM operation on page A2-79 described the architectural requirements for an EJVM at the application level. Because the EJVM is provided for use by applications, the system level description of the architecture does not require significant additional information about the EJVM.

Initialization on page A2-79 stated that, if the EJVM is compatible with the subarchitecture, the EJVM must write its required configuration to the JMCR and any other configuration registers. The EJVM must not omit this step on the assumption that the JOSCR.CV bit is 0. In other words, the EJVM must not assume that JOSCR.CV == 0, and that this will trigger entry to the Configuration Invalid handler before any bytecode instruction is executed by the Jazelle extension hardware.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

2. 新しいプロセスまたはスレッドで BXJ 命令が実行され、バイトコード命令の実行が開始された場合は、構成無効ハンドラが開始され、JOSCR.CV が 1 にセットされます。

3. 構成無効ハンドラが次の処理を実行します。
   - EJVM に必要な構成を Jazelle 構成レジスタに書き込む。
   - BXJ 命令を再試行してバイトコード命令を実行する。

これによって、Jazelle 拡張機能構成レジスタは、任意のバイトコード命令が実行される前に、関係する EJVM 用に適切にセットアップされます。構成無効ハンドラの実行中にコンテキストスイッチが発生しても正常に処理されます。

Jazelle 拡張機能へのユーザモード アクセスの監視と制御

システムでは、さまざまな方法で JOSCR.CD ビットを使用して、Jazelle 拡張機能ハードウェアへのユーザモード アクセスを監視および制御することができます。可能な使用法には、次のものが含まれます。

- OS で JOSCR.CD == 1 と JMCR.JE == 0 をセットすることによって、Jazelle 拡張機能ハードウェアへのユーザモード アクセスを防止できます。これらの設定を使用すれば、BXJ 命令が BX 命令と同じ結果になり、JMCR.JE ビットを 1 にセットする試みを含む、ハードウェアを構成するすべての試みが未定義命令例外になります。

- OS で、他のプロセスによるハードウェア使用の競合から EJVM を保護しながら、Jazelle 拡張機能ハードウェアへのユーザモード アクセスを提供する単純な機構は次のとおりです。
  - JOSCR.CD ビットを 0 にセットします。
  - コンテキストスイッチ時に JMCR を保存して復元し、新しいプロセスのためにその値を 0 に初期化します。
  - JOSCR.CV ビットは、オペレーティングシステムによるコンテキストスイッチ、または ARMv7 では例外の結果としてのコンテキストスイッチごとに 0 にセットされます。これによって、必要に応じて、EJVM によって Jazelle 拡張機能ハードウェアがその要件を満たすように再構成されることが保証されます。

  コンテキストスイッチ機構については、*Jazelle 状態の開始の制御* P. B1-79 で説明します。

EJVM の動作

アプリケーションレベルでの EJVM に関するアーキテクチャ上の要件については、*EJVM の動作* P. A2-79 で説明しました。EJVM はアプリケーションで使用するためのものであるため、アーキテクチャのシステムレベルの説明で触れるべき、EJVM の重要な追加情報はありません。

初期化 P. A2-79 には、EJVM がサブアーキテクチャと互換性がある場合に、必要な構成を JMCR とその他のすべての構成レジスタに書き込む必要があることが記載されています。EJVM では、JOSCR.CV ビットが 0 であるという前提で、この手順を省略しないで下さい。つまり、EJVM では、JOSCR.CV == 0 であり、これによって、Jazelle 拡張機能ハードウェアで任意のバイトコード命令が実行される前に構成無効ハンドラの開始がトリガされることを前提としないで下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Trivial implementation of the Jazelle extension

Jazelle direct bytecode execution support on page A2-73 introduced the possible trivial implementation of the Jazelle extension, and summarized the application level requirements of a trivial implementation. This section gives the system level description of a trivial implementation of the Jazelle extension.

A trivial implementation of the Jazelle extension must:

- Implement the JIDR with the implementer and subarchitecture fields set to zero. The register can be implemented so that the whole register is RAZ.
- Implement the JMCR as RAZ/WI.
- Implement the JOSCR either:
  - — so that it can be read and written, but its effects are ignored
  - — as RAZ/WI.

  This enables operating systems that support an EJVM to execute correctly.
- Implement the `BXJ` instruction to behave identically to the `BX` instruction in all circumstances, as required by the fact that the JMCR.JE bit is always zero. This means that Jazelle state can never be entered normally on a trivial implementation.
- Treat Jazelle state as an unsupported instruction set state, as described in *Exception return to an unsupported instruction set state* on page B1-40.

A trivial implementation does not have to extend the PC to 32 bits, that is, it can implement PC[0] as RAZ/WI. This is because the only way that PC[0] is visible in ARM or Thumb state is as a result of a processor exception occurring during Jazelle state execution, and Jazelle state execution cannot occur on a trivial implementation.

Jazelle state

All processor configuration information that can be modified by Jazelle state execution must be kept in the Application Level registers described in *ARM processor modes and core registers* on page B1-6. This ensures that the processor configuration information is preserved and restored correctly when processor exceptions and context switches occur. Configuration information can be kept either in Application Level registers or in configuration registers. In this context, configuration information is information that affects Jazelle state execution but is not modified by it.

An *Enabled Java Virtual Machine* (EJVM) implementation must check whether the implemented Jazelle extension is compatible with its use of the Application Level registers. If the implementation is compatible, the EJVM sets JE == 1 in the JMCR, see *Jazelle Main Configuration Register (JMCR)* on page A2-77. If the implementation is not compatible, the EJVM sets JE == 0 and executes without hardware acceleration.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Jazelle 拡張機能のトリビアル実装

Jazelle 直接バイトコード実行のサポート P. A2-73 では、実現可能な Jazelle 拡張機能のトリビアル実装を紹介し、トリビアル実装のアプリケーションレベルの要件について簡単に説明しました。このセクションでは、Jazelle 拡張機能のトリビアル実装について、システムレベルで説明します。

Jazelle 拡張機能のトリビアル実装では、次のことを実現する必要があります。

- 実装者フィールドとサブアーキテクチャフィールドを 0 にセットした JIDR を実装します。このレジスタは、レジスタ全体が RAZ になるように実装することができます。
- JMCR を RAZ/WI として実装します。
- 次のいずれかの方法で JOSCR を実装します。
  - – 読み書きはできるが、その影響が無視される。
  - – RAZ/WI

  これによって、EJVM をサポートするオペレーティングシステムが正常に動作します。
- JMCR.JE ビットが常に 0 であるという条件で、すべての状況で `BX` 命令と同様に動作する `BXJ` 命令を実装します。これは、トリビアル実装では、通常の方法では決して Jazelle 状態が開始されないことを意味します。
- *サポートされていない命令セット状態への例外からの復帰* P. B1-40 で説明されているように、サポートされていない命令セット状態として Jazelle 状態を扱います。

トリビアル実装では、PC を 32 ビットに拡張する必要はありません。つまり、PC[0] を RAZ/WI として実装することができます。これは、ARM 状態または Thumb 状態で PC[0] を認識可能な唯一の方法は、Jazelle 状態の実行中に発生したプロセッサ例外の結果によるものであり、トリビアル実装では Jazelle 状態の実行が起こり得ないためです。

Jazelle 状態

Jazelle 状態の実行中に変更可能なすべてのプロセッサ構成情報は、*ARM プロセッサモードとコアレジスタ* P. B1-6 で説明されているアプリケーションレベルのレジスタに保存する必要があります。これによって、プロセッサ例外やコンテキストスイッチが発生したときに、プロセッサ構成情報が保存され、正しく復元されることが保証されます。構成情報は、アプリケーションレベルのレジスタ、または構成レジスタに保存できます。この文脈における構成情報とは、Jazelle 状態の実行に影響を与えるが、それによって変更されない情報のことです。

イネーブルド Java 仮想マシン (EJVM) 実装では、実装されている Jazelle 拡張機能がアプリケーションレベルのレジスタの用途と互換性があるかどうかをチェックする必要があります。実装に互換性がある場合は、EJVM が JMCR で JE == 1 にセットします。*Jazelle メイン構成レジスタ (JMCR)* P. A2-77 を参照して下さい。実装に互換性がない場合は、EJVM が JE == 0 にセットして、ハードウェアアクセラレーションを使用せずに実行します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Jazelle state exit

The processor exits Jazelle state in IMPLEMENTATION DEFINED circumstances. Typically, this is due to attempted execution of a bytecode instruction that the implementation cannot handle in hardware, or that generates one of the Java exceptions described in Lindholm and Yellin, *The Java Virtual Machine Specification 2nd Edition*. On exit from Jazelle state, various processor registers contain SUBARCHITECTURE DEFINED values, enabling the EJVM to resume software execution of the bytecode program correctly.

The processor also exits Jazelle state when a processor exception occurs. The CPSR is copied to the banked SPSR for the exception mode, so the banked SPSR contains J == 1 and T == 0, and Jazelle state is restored on return from the exception when the SPSR is copied back into the CPSR. With the restriction that Jazelle state execution can modify only Application Level registers, this ensures that all registers are correctly preserved and can be restored by the exception handlers. Configuration and control registers can be modified in the exception handler itself as described in *Jazelle state configuration and control* on page B1-77 and *Jazelle OS Control Register (JOSCR)* on page B1-77.

Specific considerations apply to processor exceptions, see *Exception handling in the Jazelle extension* on page B1-74.

It is IMPLEMENTATION DEFINED whether Jazelle extension hardware contains state that is both:

- modified during Jazelle state execution
- held outside the Application Level registers during Jazelle state execution.

If such state exists, the implementation must:

- Initialize the state from one or more of the Application Level registers whenever Jazelle state is entered, whether as the result of:
  - — the execution of a BXJ instruction
  - — returning from a processor exception.
- Write the state into one or more of the Application Level registers whenever Jazelle state is exited, whether as a result of taking a processor exception or of IMPLEMENTATION DEFINED circumstances.
- Ensure that the mechanism for writing the state into Application Level registers on taking a processor exception, and initializing the state from Application Level registers on returning from that exception, ensures that the state is correctly preserved and restored over the exception.

Additional Jazelle state restrictions

The Jazelle extension hardware must obey the following restrictions:

- It must not change processor mode other than by taking one of the standard ARM processor exceptions.
- It must not access banked versions of registers other than the ones belonging to the processor mode in which it is entered.
- It must not do anything that is illegal for an UNPREDICTABLE instruction. That is, it must not:
  - — generate a security loophole
  - — halt or hang the processor or any other part of the system.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

Jazelle 状態の終了

プロセッサは、実装定義の状況で Jazelle 状態を終了します。これは通常、実装のハードウェアで処理できないバイトコード命令、または、Lindholm 氏と Yellin 氏による『*The Java Virtual Machine Specification 2nd Edition*』内の Java 例外の 1 つを生成するバイトコード命令を実行しようとしたことが原因です。Jazelle 状態の終了時に、さまざまなプロセッサレジスタにサブアーキテクチャ定義の値が保存されるため、EJVM は、バイトコードプログラムのソフトウェア実行を正常に再開することができます。

プロセッサは、プロセッサ例外が発生したときにも Jazelle 状態を終了します。CPSR は例外モードに対応するバンク SPSR にコピーされるため、バンク SPSR には J == 1 と T == 0 が含まれており、例外からの復帰時に SPSR が CPSR に戻されると Jazelle 状態が復元されます。Jazelle 状態の実行ではアプリケーションレベルのレジスタしか変更できないという制約によって、すべてのレジスタが正しく保存され、例外ハンドラによって復元されることが保証されます。*Jazelle 状態の構成と制御* P. B1-77 と *Jazelle OS 制御レジスタ (JOSCR)* P. B1-77 で説明されているように、構成レジスタと制御レジスタは例外ハンドラ自体で変更することができます。

特定の考慮事項がプロセッサ例外に適用されます。*Jazelle 拡張機能での例外処理* P. B1-74 を参照して下さい。

Jazelle 拡張機能ハードウェアに、次の両方に該当する状態を含めるかどうかは実装定義です。

- Jazelle 状態の実行中に変更された状態
- Jazelle 状態の実行中にアプリケーションレベル レジスタ以外に保存された状態

このような状態が存在する場合、実装で次の動作のことを実行する必要があります。

- 次のいずれかの結果によって Jazelle 状態が開始した場合は、常に 1 つ以上のアプリケーションレベルのレジスタから状態を初期化します。
  - – BXJ 命令の実行
  - – プロセッサ例外からの復帰
- プロセッサ例外が取得された結果、または実装定義の状況の結果として Jazelle 状態が終了したときは、常に 1 つ以上のアプリケーションレベルのレジスタに状態を書き込みます。
- プロセッサ例外の取得時に状態をアプリケーションレベルのレジスタに書き込み、その例外からの復帰時にアプリケーションレベルのレジスタから状態を初期化する機構によって、例外の前後で状態が正しく保存され、復元されることを保証します。

その他の Jazelle 状態の制限

Jazelle 拡張機能ハードウェアは、次の制約に従う必要があります。

- 標準の ARM プロセッサ例外のいずれかを取得すること以外でプロセッサモードを変更しないこと。
- 開始されたときのプロセッサモードに属していないバンクバージョンのレジスタにアクセスしないこと。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

As a result of these requirements, Jazelle state can be entered from User mode without risking a breach of OS security.

In addition, Jazelle state execution is UNPREDICTABLE in FIQ mode.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- 予測不能な命令について、規則に反した動作を行わないこと。つまり、次の動作を行わないこと。
  - セキュリティの抜け道を生成する。
  - プロセッサまたはシステムのその他の部分を停止またはハングする。

これらの要件を満たすことによって、Jazelle 状態は、OS のセキュリティに破綻が生じる危険なしに、ユーザモードから開始することができます。

また、FIQ モードでは、Jazelle 状態の実行は予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter B2
Common Memory System Architecture Features

This chapter provides a system-level view of the general features of the memory system. It contains the following sections:

- *About the memory system architecture* on page B2-2
- *Caches* on page B2-3
- *Implementation defined memory system features* on page B2-27
- *Pseudocode details of general memory system operations* on page B2-29.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 B2 章
一般的なメモリシステム アーキテクチャの機能

本章では、システムレベルから見たメモリシステムの一般的な機能について説明します。本章は次のセクションから構成されています。

- *メモリシステム アーキテクチャについて* P. B2-2
- キャッシュ P. B2-3
- *実装定義のメモリシステムの機能* P. B2-27
- *一般的なメモリシステム操作の擬似コードの詳細* P. B2-29

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.1 About the memory system architecture

The ARM architecture supports different implementation choices for the memory system microarchitecture and memory hierarchy, depending on the requirements of the system being implemented. In this respect, the memory system architecture describes a design space in which an implementation is made. The architecture does not prescribe a particular form for the memory systems. Key concepts are abstracted in a way that enables implementation choices to be made while enabling the development of common software routines that do not have to be specific to a particular microarchitectural form of the memory system. For more information about the concept of a hierarchical memory system see *Memory hierarchy* on page A3-52.

B2.1.1 Form of the memory system architecture

ARMv7 supports different forms of the memory system architecture, that map onto the different architecture profiles. Two of these are described in this manual:

- ARMv7-A, the A profile, requires the inclusion of a *Virtual Memory System Architecture* (VMSA), as described in Chapter B3 *Virtual Memory System Architecture (VMSA)*.
- ARMv7-R, the R profile, requires the inclusion of a *Protected Memory System Architecture* (PMSA), as described in Chapter B4 *Protected Memory System Architecture (PMSA)*.

Both of these memory system architectures provide mechanisms to split memory into different regions. Each region has specific memory types and attributes. The two memory system architectures have different capabilities and programmers' models.

The memory system architecture model required by ARMv7-M, the M profile, is outside the scope of this manual. It is described in the *ARMv7-M Architecture Reference Manual*.

B2.1.2 Memory attributes

Summary of ARMv7 memory attributes on page A3-25 summarizes the memory attributes, including how different memory types have different attributes. Each region of memory has a set of memory attributes:

- in a PMSA implementation the attributes are part of each MPU memory region definition
- in a VMSA implementation the translation table entry that defines a virtual memory region also defines the attributes for that region.

B2.1.3 Levels of cache

From ARMv7, the architecturally-defined cache control mechanism covers multiple levels of cache, as described in *Caches* on page B2-3. Also, it permits levels of cache beyond the scope of these cache control mechanisms, see *System-level caches* on page B2-26.

Note

Before ARMv7, the architecturally-defined cache control mechanism covers only a single level of cache, and any support for other levels of cache is IMPLEMENTATION DEFINED.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B2.1 メモリシステム アーキテクチャについて

ARM アーキテクチャでは、メモリシステム マイクロアーキテクチャとメモリ階層に関して、実装対象システムの要件に応じたさまざまな実装の選択肢が用意されています。この観点では、メモリシステム アーキテクチャとは、実装を具体化するための設計空間を意味します。アーキテクチャによって、特定のメモリシステムの形式が規定されるわけではありません。主要な概念は、特定のマイクロアーキテクチャ形式のメモリシステムに特化する必要のない汎用的なソフトウェアルーチンの開発を可能にしながら、実装の選択を行うことができるような方法で抽象化されています。階層メモリシステムの概念の詳細については、*メモリの階層* P. A3-52 を参照して下さい。

B2.1.1 メモリシステム アーキテクチャの形式

ARMv7 では、各種のアーキテクチャプロファイルにマッピングされた、さまざまな形式のメモリシステム アーキテクチャがサポートされています。本書では、これらのうちの 2 つのアーキテクチャについて説明します。

- ARMv7-A（A プロファイル）では、第 B3 章 *「仮想メモリシステム アーキテクチャ (VMSA)」* で説明されているように*仮想メモリシステム アーキテクチャ*(VMSA) を組み込む必要があります。
- ARMv7-R（R プロファイル）では、第 B4 章 *「保護メモリシステム アーキテクチャ (PMSA)」* で説明されているように*保護メモリシステム アーキテクチャ*(PMSA) を組み込む必要があります。

これらのメモリシステム アーキテクチャはいずれも、メモリを複数の領域に分割する機構を提供します。各領域には、特定のメモリタイプや属性が付与されます。この 2 つのメモリシステム アーキテクチャは、機能とプログラマモデルに違いがあります。

ARMv7-M（M プロファイル）に必要なメモリシステム アーキテクチャモデルに関する説明は本書の範囲外です。これについては、『*ARMv7-M アーキテクチャ リファレンスマニュアル*』を参照して下さい。

B2.1.2 メモリ属性

それぞれのメモリタイプにどのような属性があるかを含め、メモリ属性の概要を *ARMv7 のメモリ属性の要約* P. A3-25 で説明します。各メモリ領域には、次のような一連のメモリ属性が設定されます。

- PMSA 実装では、属性が各 MPU メモリ領域定義の一部になっています。
- VMSA 実装では、仮想メモリ領域を定義する変換テーブルのエントリによって、その領域の属性も定義されます。

B2.1.3 キャッシュレベル

キャッシュ P. B2-3 で説明されているように、ARMv7 およびそれ以降のアーキテクチャ定義のキャッシュ制御機構では、複数のキャッシュレベルがカバーされています。また、これらのキャッシュ制御機構の範囲を超えたキャッシュレベルを使用することもできます。*システムレベルのキャッシュ* P. B2-26 を参照して下さい。

注

ARMv7 以前のアーキテクチャ定義のキャッシュ制御機構では、単一のキャッシュレベルしかカバーされず、その他のキャッシュレベルのサポートは実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.2 Caches

The concept of caches is described in *Caches and memory hierarchy* on page A3-51. This section describes the cache identification and control mechanisms in ARMv7. These are described in the following sections:

- *Cache identification*
- *Cache behavior* on page B2-5
- *Cache enabling and disabling* on page B2-8
- *Cache maintenance functionality* on page B2-9
- *The interaction of cache lockdown with cache maintenance* on page B2-18
- *Branch predictors* on page B2-19
- *Ordering of cache and branch predictor maintenance operations* on page B2-21
- *Multiprocessor effects on cache maintenance operations* on page B2-23
- *System-level caches* on page B2-26.

Note

The cache identification and control mechanisms for previous versions of the ARM architecture are described in:

- *Cache support* on page AppxG-21, for ARMv6
- *Cache support* on page AppxH-21, for the ARMv4 and ARMv5 architectures.

B2.2.1 Cache identification

The ARMv7 cache identification consists of a set of registers that describe the implemented caches that are under the control of the processor:

- A single Cache Type Register defines:
  - the minimum line length of any of the instruction caches
  - the minimum line length of any of the data or unified caches
  - the cache indexing and tagging policy of the Level 1 instruction cache.

  For more information, see:
  - *c0, Cache Type Register (CTR)* on page B3-83, for a VMSA implementation
  - *c0, Cache Type Register (CTR)* on page B4-34, for a PMSA implementation.
- A single Cache Level ID Register defines:
  - the type of cache implemented at a each cache level, up to the maximum of seven levels
  - the Level of Coherence for the caches
  - the Level of Unification for the caches.

  For more information, see:
  - *c0, Cache Level ID Register (CLIDR)* on page B3-92, for a VMSA implementation
  - *c0, Cache Level ID Register (CLIDR)* on page B4-41, for a PMSA implementation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.2 キャッシュ

キャッシュの概念については、*キャッシュとメモリ階層* P. A3-51 を参照して下さい。このセクションでは、ARMv7 におけるキャッシュの識別および制御機構について説明します。この説明は、次の各サブセクションで構成されています。

- *キャッシュの識別*
- *キャッシュの動作* P. B2-5
- *キャッシュの稼動と非稼動* P. B2-8
- *キャッシュ保守機能* P. B2-9
- *キャッシュロックダウンとキャッシュ保守との相互動作* P. B2-18
- *分岐予測器* P. B2-19
- *キャッシュの保守操作と分岐予測器の保守操作との順序付け* P. B2-21
- *キャッシュの保守操作に対するマルチプロセッサの影響* P. B2-23
- *システムレベルのキャッシュ* P. B2-26

注

以前のバージョンの ARM アーキテクチャに関するキャッシュの識別および制御機構については、次のセクションを参照して下さい。

- *キャッシュのサポート* P. 付録 G-21（ARMv6 の場合）
- *キャッシュのサポート* P. 付録 H-21（ARMv4 および ARMv5 アーキテクチャの場合）

B2.2.1 キャッシュの識別

ARMv7 のキャッシュの識別は、プロセッサの制御下にある実装済みのキャッシュを記述する一連のレジスタで構成されます。

- 単一のキャッシュタイプ レジスタによって、次のものが定義されます。
  - 命令キャッシュの最小ライン長
  - データキャッシュまたは統一キャッシュの最小ライン長
  - レベル 1 命令キャッシュの索引付けおよびタグ付けポリシー

  詳細については、次の各セクションを参照して下さい。
  - *c0、キャッシュタイプ レジスタ (CTR)* P. B3-83（VMSA 実装の場合）
  - *c0、キャッシュタイプ レジスタ (CTR)* P. B4-34（PMSA 実装の場合）
- 単一のキャッシュレベル ID レジスタによって、次のものが定義されます。
  - 各キャッシュレベルに実装されるキャッシュのタイプ、最大 7 レベルまで
  - キャッシュのコヒーレンシレベル
  - キャッシュの統一レベル

  詳細については、次の各セクションを参照して下さい。
  - *c0、キャッシュレベル ID レジスタ (CLIDR)* P. B3-92（VMSA 実装の場合）
  - *c0、キャッシュレベル ID レジスタ (CLIDR)* P. B4-41（PMSA 実装の場合）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- A single Cache Size Selection Register selects the cache level and cache type of the current Cache Size Identification Register, see:
  - *c0, Cache Size Selection Register (CSSELR)* on page B3-95, for a VMSA implementation
  - *c0, Cache Size Selection Register (CSSELR)* on page B4-43, for a PMSA implementation.
- For each implemented cache, across all the levels of caching, a Cache Size Identification Register defines:
  - whether the cache supports Write-Through, Write-Back, Read-Allocate and Write-Allocate
  - the number of sets, associativity and line size of the cache.

  For more information, see:
  - *c0, Cache Size ID Registers (CCSIDR)* on page B3-91, for a VMSA implementation
  - *c0, Cache Size ID Registers (CCSIDR)* on page B4-40, for a PMSA implementation.

Identifying the cache resources in ARMv7

From ARMv7 the architecture defines support for multiple levels of cache, up to a maximum of seven levels. This means the process of identifying the cache resources available to the processor in an ARMv7 implementation is more complicated. To obtain this information:

1. Read the Cache Type Register to find the indexing and tagging policy used for the Level 1 instruction cache. This register also provides the size of the smallest cache lines used for the instruction caches, and for the data and unified caches. These values are used in cache maintenance operations.
2. Read the Cache Level ID Register to find what caches are implemented. The register includes seven Cache type fields, for cache levels 1 to 8. Scanning these fields, starting from Level 1, identifies the instruction, data or unified caches implemented at each level. This scan ends when it reaches a level at which no caches are defined. The Cache Level ID Register also provides the Level of Unification and the Level of Coherency for the cache implementation.
3. For each cache identified at stage 2:
   - Write to the Cache Size Selection Register to select the required cache. A cache is identified by its level, and whether it is:
     - an instruction cache
     - a data or unified cache.
   - Read the Cache Size ID Register to find details of the cache.

Note

In ARMv6, only the Level 1 caches are architecturally defined, and the Cache Type Register holds details of the caches. For more information, see *Cache support* on page AppxG-21.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- 単一のキャッシュサイズ選択レジスタによって、現在のキャッシュサイズ識別レジスタのキャッシュレベルとキャッシュタイプが選択されます。次のセクションを参照して下さい。
  - *c0、キャッシュサイズ選択レジスタ (CSSELR)* P. B3-95（VMSA 実装の場合）
  - *c0、キャッシュサイズ選択レジスタ (CSSELR)* P. B4-43（PMSA 実装の場合）
- 実装済みの各キャッシュについて、すべてのキャッシュレベルにわたり、キャッシュサイズ識別レジスタによって次のものが定義されます。
  - キャッシュでライトスルー、ライトバック、読み出し割り当て、および書き込み割り当てがサポートされているかどうか。
  - キャッシュのセット数、アソシエティビティ、およびラインサイズ

  詳細については、次の各セクションを参照して下さい。
  - *c0、キャッシュサイズ ID レジスタ (CCSIDR)* P. B3-91（VMSA 実装の場合）
  - *c0、キャッシュサイズ ID レジスタ (CCSIDR)* P. B4-40（PMSA 実装の場合）

ARMv7 におけるキャッシュリソースの識別

ARMv7 およびそれ以降では、最大 7 レベルまでの複数キャッシュレベルのサポートがアーキテクチャで定義されています。これは、ARMv7 実装のプロセッサで使用可能なキャッシュリソースの識別処理がより複雑になることを意味します。この情報を得るには、次の手順を使用します。

1. キャッシュタイプ レジスタを読み出して、レベル 1 命令キャッシュで使用されている索引付けおよびタグ付けポリシーを特定します。このレジスタでは、命令キャッシュ、データキャッシュ、および統一キャッシュで使用されるキャッシュラインの最小サイズも提供されます。これらの値は、キャッシュ保守操作で使用されます。
2. キャッシュレベル ID レジスタを読み出して、実装されているキャッシュを特定します。このレジスタには、キャッシュレベルの 1 ～ 8 に対応する 7 つのキャッシュタイプ フィールドが含まれています。レベル 1 から順にこれらのフィールドをスキャンすることによって、各レベルの命令キャッシュ、データキャッシュ、または統一キャッシュが識別されます。このスキャンは、キャッシュが定義されていないレベルに到達した時点で終了します。キャッシュレベル ID レジスタでは、キャッシュ実装の統一レベルとコヒーレンシレベルも提供されます。
3. ステージ 2 で識別された各キャッシュについて、次の操作を実行します。
   - キャッシュサイズ選択レジスタに書き込みを行って、必要なキャッシュを選択します。キャッシュはレベルと、次のどちらであるかによって識別されます。
     - 命令キャッシュ
     - データキャッシュまたは統一キャッシュ
   - キャッシュサイズ ID レジスタを読み出して、キャッシュの詳細を特定します。

注

ARMv6 では、レベル 1 キャッシュのみがアーキテクチャ定義で、キャッシュタイプ レジスタにキャッシュの詳細が保持されます。詳細については、*キャッシュのサポート* P. 付録 G-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.2.2 Cache behavior

The behavior of caches in an ARMv7 implementation is summarized in the following subsections:

- *General behavior of the caches*
- *Behavior of the caches at reset* on page B2-6
- *Behavior of Preload Data (PLD, PLDW) and Preload Instruction (PLI) with caches* on page B2-7.

General behavior of the caches

When a memory location is marked with a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache still depends on many aspects of the implementation. Typically, the following non-exhaustive list of factors might be involved:

- the size, line-length, and associativity of the cache
- the cache allocation algorithm
- activity by other elements of the system that can access the memory
- instruction prefetching algorithms
- data prefetching algorithms
- interrupt behaviors.

Given this range of factors, and the large variety of cache systems that might be implemented, the architecture cannot guarantee whether:

- a memory location present in the cache remains in the cache
- a memory location not present in the cache is brought into the cache.

Instead, the following principles apply to the behavior of caches:

- The architecture has a concept of an entry locked down in the cache. How lockdown is achieved is IMPLEMENTATION DEFINED, and lockdown might not be supported by:
  - — a particular implementation
  - — some memory attributes.
- An unlocked entry in the cache cannot be relied upon to remain in the cache. If an unlocked entry does remain in the cache, it cannot be relied upon to remain incoherent with the rest of memory. In other words, software must not assume that an unlocked item that remains in the cache remains dirty.
- A locked entry in the cache can be relied upon to remain in the cache. A locked entry in the cache cannot be relied upon to remain incoherent with the rest of memory, that is, it cannot be relied on to remain dirty.

  Note

  For more information, see *The interaction of cache lockdown with cache maintenance* on page B2-18.

- If a memory location is marked as Cacheable there is no mechanism by which it can be guaranteed not to be allocated to an enabled cache at any time. Any application must assume that any Cacheable memory location can be allocated to any enabled cache at any time.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.2.2 キャッシュの動作

ARMv7 実装におけるキャッシュの動作について、次のサブセションで説明します。

- *一般的なキャッシュの動作*
- *リセット時のキャッシュの動作* P. B2- 6
- *キャッシュを使用したデータのプリロード (PLD、PLDW) と命令のプリロード (PLI) の動作* P. B2- 7

一般的なキャッシュの動作

メモリ位置がノーマルキャッシュ可能メモリ属性としてマークされている場合、メモリ位置のコピーがまだキャッシュ内に保持されているかどうかの判定は、実装の多くの側面によって異なります。通常は、次のような要因が関係します。ただし、要因はこのリストのものですべてではありません。

- キャッシュのサイズ、ライン長、アソシエティビティ
- キャッシュ割り当てアルゴリズム
- メモリにアクセス可能なシステムの他のエレメントによる動作
- 命令プリフェッチアルゴリズム
- データプリフェッチ アルゴリズム
- 割り込み動作

このように広範な要因があり、さらに実装される可能性があるキャッシュシステムには多くの種類が存在するため、アーキテクチャで次の条件を保証することはできません。

- キャッシュに存在するメモリ位置がキャッシュに保持される。
- キャッシュに存在しないメモリ位置がキャッシュに読み込まれる。

代わりに、キャッシュの動作には次の原則が適用されます。

- アーキテクチャには、キャッシュ内でロックダウンされるエントリの概念が含まれています。ロックダウンの実現方法は実装定義であり、次の条件によりロックダウンがサポートされない場合があります。
  - – 特定の実装
  - – 一部のメモリ属性
- キャッシュのロックされていないエントリは、キャッシュに残ることを前提とできません。ロックされていないエントリがキャッシュに残っている場合、他のメモリと不整合な状態のままであることを前提とできません。つまり、ソフトウェアでは、キャッシュに残っているロックされていない項目がダーティなままであることを前提としないで下さい。
- キャッシュのロックされたエントリは、キャッシュに残ることを前提とできます。キャッシュのロックされたエントリは、他のメモリと不整合なまま、つまりダーティなままであることを前提とできません。

  注

  詳細については、*キャッシュロックダウンとキャッシュ保守との相互動作* P. B2- 18 を参照して下さい。

- メモリ位置がキャッシュ可能としてマークされている場合、その位置がある時点で許可されたキャッシュに割り当てられていないことを保証できる機構は存在しません。アプリケーションでは、キャッシュ可能なメモリ位置はいつでも、任意の許可されたキャッシュに割り当てられる可能性があることを前提とする必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- If the cache is disabled, it is guaranteed that no new allocation of memory locations into the cache will occur.
- If the cache is enabled, it is guaranteed that no memory location that does not have a Cacheable attribute is allocated into the cache.
- If the cache is enabled, it is guaranteed that no memory location is allocated to the cache if its translation table attributes or region attributes prevent privileged read access.
- Any memory location that is marked as Normal Shareable is guaranteed to be coherent with all masters in that shareability domain for data accesses.
- Any memory location is not guaranteed to remain incoherent with the rest of memory.
- The eviction of a cache entry from a cache level can overwrite memory that has been written by another observer only if the entry contains a memory location that has been written to by a processor that controls that cache. The maximum size of the memory that can be overwritten is called the *Cache Writeback Granule*. In some implementations the CTR identifies the Cache Writeback Granule, see:
  — *c0, Cache Type Register (CTR)* on page B3-83 for a VMSA implementation
  — *c0, Cache Type Register (CTR)* on page B4-34 for a PMSA implementation.
- The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer, if it had previously been visible to that observer.

For the purpose of these principles, a cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to its size.

In addition, in ARMv7, in the following situations it is UNPREDICTABLE whether the location is returned from cache or from memory:

- The location is not marked as Cacheable but is contained in the cache. This situation can occur if a location is marked as Non-cacheable after it has been allocated into the cache.
- The location is marked as Cacheable and might be contained in the cache, but the cache is disabled.

Behavior of the caches at reset

In ARMv7:

- All caches are disabled at reset.
- An implementation can require the use of a specific cache initialization routine to invalidate its storage array before it is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, but the routine must be documented clearly as part of the documentation of the device.
- It is IMPLEMENTATION DEFINED whether an access can generate a cache hit when the cache is disabled. If an implementation permits cache hits when the cache is disabled the cache initialization routine must:
  — provide a mechanism to ensure the correct initialization of the caches
  — be documented clearly as part of the documentation of the device.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- キャッシュが非稼動の場合、キャッシュに対する新しいメモリ位置の割り当てが行われないことが保証されます。
- キャッシュが稼動している場合、キャッシュ可能属性を持たないメモリ位置がキャッシュに割り当てられないことが保証されます。
- キャッシュが稼動している場合、その変換テーブル属性または領域属性によって特権読み出しアクセスが拒否されていれば、メモリ位置はキャッシュに割り当てられないことが保証されます。
- ノーマル共有可としてマークされたメモリ位置は、そのデータアクセスの共有可ドメイン内のすべてのマスタと整合性が保証されます。
- すべてのメモリ位置は、他のメモリと不整合のままであることが保証されません。
- キャッシュレベルからキャッシュエントリを退出する場合、そのエントリに、そのキャッシュを制御しているプロセッサから書き込まれたメモリ位置が含まれている場合にのみ、他のオブザーバから書き込まれたメモリを上書きすることができます。上書き可能なメモリの最大サイズは、*キャッシュライトバックの粒度*と呼ばれます。一部の実装では、CTR によってキャッシュライトバックの粒度が識別されます。次のセクションを参照して下さい。
  – *c0、キャッシュタイプ レジスタ (CTR)* P. B3-83（VMSA 実装の場合）
  – *c0、キャッシュタイプ レジスタ (CTR)* P. B4-34 （PMSA 実装の場合）
- キャッシュに対するメモリ位置の割り当てでは、そのメモリ位置が以前にオブザーバに対して可視であった場合、メモリ位置の最新の値をそのオブザーバに可視でないようにすることはできません。

これらの原則を満たすために、1 つのキャッシュエントリは、16 バイト以上 2KB 以下で、サイズに応じてアラインされた連続するアドレス空間をカバーします。

また、ARMv7 では、次の状況において、特定の位置がキャッシュとメモリのどちらから返されるかは予測不能です。

- その位置がキャッシュ可能としてマークされていないが、キャッシュに存在している場合。この状況は、その位置がキャッシュに割り当てられた後でキャッシュ不可としてマークされた場合に発生する可能性があります。
- その位置がキャッシュ可能としてマークされており、キャッシュに存在している可能性があるが、キャッシュが非稼動の場合。

リセット時のキャッシュの動作

ARMv7 では、次の動作が行われます。

- すべてのキャッシュはリセット時に非稼動になります。
- 実装では、特定のキャッシュ初期化ルーチンを使用して、キャッシュを稼動させる前にその記憶域アレイを無効にすることが必要な場合があります。必要な初期化ルーチンの詳細な形式は実装定義ですが、このルーチンはデバイスのドキュメントの一部として明示的に文書化する必要があります。
- キャッシュが非稼動のとき、アクセスによってキャッシュヒットが生成できるかどうかは実装定義です。実装で、キャッシュが非稼動のときにキャッシュヒットが許可されている場合は、キャッシュ初期化ルーチンについて次の作業を行う必要があります。
  – キャッシュの正確な初期化を保証するための機構を提供する。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In particular, if an implementation permits cache hits when the cache is disabled and the cache contents are not invalidated at reset, the initialization routine must avoid any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.

- ARM recommends that whenever an invalidation routine is required, it is based on the ARMv7 cache maintenance operations.

When they are enabled the state of the caches is UNPREDICTABLE if the appropriate initialization routine has not been performed.

Similar rules apply:
- to branch predictor behavior, see *Behavior of the branch predictors at reset* on page B2-21
- on an ARMv7-A implementation, to TLB behavior, see *TLB behavior at reset* on page B3-55.

Note

Before ARMv7, caches are invalidated by the assertion of reset, see *Cache behavior at reset* on page AppxG-23.

Behavior of Preload Data (PLD, PLDW) and Preload Instruction (PLI) with caches

Preload Data and Preload Instruction operations are provided by the `PLD` and `PLI` instructions. These are implemented in the ARM and Thumb instruction sets. The Multiprocessing Extensions add the `PLDW` instruction.

`PLD`, `PLDW` and `PLI` act as hints to the memory system, and as such their operation does not cause a precise abort to occur. However, a memory operation performed as a result of one of these memory system hints might trigger an asynchronous event, so influencing the execution of the processor. Examples of the asynchronous events that might be triggered are asynchronous aborts and interrupts.

A `PLD` or `PLDW` instruction is guaranteed not to cause any effect to the caches, or TLB, or memory other than the effects that, for permission or other reasons, the equivalent load from the same location with the same context and at the same privilege level can cause.

A `PLD` or `PLDW` instruction is guaranteed not to access Strongly-ordered or Device memory.

A `PLI` instruction is guaranteed not to cause any effect to the caches, or TLB, or memory other than the effects that, for permission or other reasons, the fetch resulting from changing the PC to the location specified by the `PLI` instruction with the same context and at the same privilege level can cause.

A `PLI` instruction is guaranteed not to access Strongly-ordered or Device memory. In a VMSA implementation, a `PLI` instruction must not perform any accesses when the MMU is disabled.

Note

In ARMv6, an instruction prefetch is provided by the optional Prefetch instruction cache line operation in CP15 c7, with encoding `<opc1>` == 0, `<CRm>` == c13, `<opc2>` == 1, see *c7, Cache operations* on page AppxG-38.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

– デバイスの説明書の一部として明示的に文書化する。

特に、キャッシュが非稼動の間のキャッシュヒットが実装で許可されており、リセット時にキャッシュの内容が無効にならない場合は、初期化ルーチンは、初期化されていないキャッシュから実行が行われる可能性を排除する必要があります。初期化ルーチンで、固定の命令シーケンスを限定された範囲のメモリに配置するように要求することができます。

· 無効化ルーチンが必要な場合は、ARMv7 のキャッシュ保守操作に基づくことをお勧めします。

適切な初期化ルーチンが実行されなかった場合、キャッシュが稼動されたときの状態は予測不能です。

次の動作にも、同様の規則が適用されます。
· 分岐予測器の動作（*リセット時の分岐予測器の動作* P. B2-21 を参照）
· ARMv7-A 実装上の TLB の動作（*リセット時の TLB の動作* P. B3-55 を参照）

注

ARMv7 以前は、リセットをアサートすることによってキャッシュが無効化されます。*リセット時のキャッシュの動作* P. 付録 G-23 を参照して下さい。

キャッシュを使用したデータのプリロード (PLD、PLDW) と命令のプリロード (PLI) の動作

データのプリロードと命令のプリロードは、`PLD` 命令と `PLI` 命令によって提供されます。これら 2 つの命令は、ARM 命令セットと Thumb 命令セットに実装されています。`PLDW` 命令は、マルチプロセッシング拡張機能で追加されます。

`PLD`、`PLDW`、`PLI` は、メモリシステムに対するヒントとして機能するため、正確なアボートが発生しません。ただし、これらのメモリシステム ヒントのいずれかの結果として実行されるメモリ操作によって、非同期イベントがトリガされる可能性があるため、プロセッサの実行に影響します。トリガされる可能性のある非同期イベントの例は、非同期アボートと非同期割り込みです。

`PLD` 命令または `PLDW` 命令は、アクセス許可またはその他の理由によって、同じ特権レベルで同じコンテキストを含む同じ場所からの同等のロードによる影響以外に、キャッシュ、TLB、またはメモリに影響を与えないことが保証されています。

`PLD` 命令または `PLDW` 命令は、ストロングリオーダ メモリまたはデバイスメモリにアクセスしないことが保証されています。

`PLI` 命令は、アクセス許可またはその他の理由によって、同じ特権レベルで同じコンテキストを含む `PLI` 命令で指定された場所に PC が変更されたことにより発生するフェッチによる影響以外に、キャッシュ、TLB、またはメモリに影響を与えないことが保証されています。

`PLI` 命令は、ストロングリオーダ メモリまたはデバイスメモリにアクセスしないことが保証されています。VMSA 実装で、MMU が非稼動の場合は、`PLI` 命令でアクセスを実行しないで下さい。

注

ARMv6 では、`<opc1>` == 0、`<CRm>` == c13、`<opc2>` == 1 のエンコードを使用した CP15 c7 のオプションのプリフェッチ命令キャッシュライン操作によって、命令のプリフェッチが提供されます。*c7、キャッシュ操作* P. 付録 G-38 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Cache lockdown

Cache lockdown requirements can conflict with the management of hardware coherency. For this reason, ARMv7 introduces significant changes in this area, compared to previous versions of the ARM architecture. These changes recognize that, in many systems, cache lockdown is inappropriate.

For an ARMv7 implementation:

- There is no requirement to support cache lockdown.
- If cache lockdown is supported, the lockdown mechanism is IMPLEMENTATION DEFINED. However key properties of the interaction of lockdown with the architecture must be described in the implementation documentation.
- The Cache Type Register does not hold information about lockdown. This is a change from ARMv6. However some CP15 c9 encodings are available for IMPLEMENTATION DEFINED, cache lockdown features, see *Implementation defined memory system features* on page B2-27.

Note

For details of cache lockdown in ARMv6 see *c9, Cache lockdown support* on page AppxG-45.

B2.2.3 Cache enabling and disabling

Levels of cache on page B2-2 indicates that:

- from ARMv7 the architecture defines the control of multiple levels of cache
- before ARMv7 the architecture defines the control of only one level of cache.

This means the mechanism for cache enabling and disabling caches changes in ARMv7. In both cases, enabling and disabling of caches is controlled by the SCTLR.C and SCTLR.I bits, see:

- *c1, System Control Register (SCTLR)* on page B3-96, for a VMSA implementation
- *c1, System Control Register (SCTLR)* on page B4-45, for a PMSA implementation.

In ARMv7:

- The SCTLR.C bit enables or disables all data and unified caches, across all levels of cache visible to the processor.
- The SCTLR.I bit enables or disables all instruction caches, across all levels of cache visible to the processor.
- If an implementation requires finer-grained control of cache enabling it can implement control bits in the Auxiliary Control Register for this purpose. For example, an implementation might define control bits to enable and disable the caches at a particular level. For more information about the Auxiliary Control Register see:
  - *c1, Implementation defined Auxiliary Control Register (ACTLR)* on page B3-103, for a VMSA implementation
  - *c1, Implementation defined Auxiliary Control Register (ACTLR)* on page B4-50, for a PMSA implementation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

 抄訳

キャッシュロックダウン

キャッシュロックダウン要件は、ハードウェアコヒーレンシの管理と競合する可能性があります。そのため ARMv7 では、この領域が、以前のバージョンの ARM アーキテクチャに比べて大幅に変更されています。これらの変更は、多くのシステムでキャッシュロックダウンが適切でないことを認識しています。

ARMv7 実装の場合、次の規則が適用されます。

- キャッシュロックダウンをサポートする要件はありません。
- キャッシュロックダウンをサポートする場合、ロックダウン機構は実装定義です。ただし、ロックダウンとアーキテクチャとの相互動作の主な特徴を実装の説明書に記載する必要があります。
- キャッシュタイプ レジスタには、ロックダウンに関する情報が保持されません。これは、ARMv6 から変更されたものです。ただし、一部の CP15 c9 エンコードを実装定義のキャッシュロックダウン機能に使用できます。*実装定義のメモリシステムの機能* P. B2-27 を参照して下さい。

注

ARMv6 でのキャッシュロックダウンの詳細については、*c9、キャッシュロックダウンのサポート* P. 付録 G-45 を参照して下さい。

B2.2.3 キャッシュの稼動と非稼動

キャッシュレベル P. B2-2 は、次のことを示しています。

- ARMv7 およびそれ以降では、アーキテクチャで複数のキャッシュレベルの制御が定義されています。
- ARMv7 以前では、アーキテクチャで単一のキャッシュレベルの制御のみが定義されています。

これは、キャッシュの稼動と非稼動の機構が ARMv7 で変更されたことを意味します。どちらの場合も、キャッシュの稼動と非稼動は、SCTLR.C ビットと SCTLR.I ビットによって制御されます。次のセクションを参照して下さい。

- *c1、システム制御レジスタ (SCTLR)* P. B3-96（VMSA 実装の場合）
- *c1、システム制御レジスタ (SCTLR)* P. B4-45（PMSA 実装の場合）

ARMv7 では、次の規則が適用されます。

- SCTLR.C ビットは、プロセッサから可視のすべてのキャッシュレベルにわたって、すべてのデータキャッシュと統一キャッシュを稼動または非稼動にします。
- SCTLR.I ビットは、プロセッサから可視のすべてのキャッシュレベルにわたって、すべての命令キャッシュを稼動または非稼動にします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

In ARMv6, the SCTLR I, C, and W bits provide separate enables for the level 1 instruction cache (if implemented), the level 1 data or unified cache, and write buffering. For more information, see *c1, System Control Register (SCTLR)* on page AppxG-34.

When a cache is disabled:

- it is IMPLEMENTATION DEFINED whether a cache hit occurs if a location that is held in the cache is accessed
- any location that is not held in the cache is not brought into the cache as a result of a memory access.

The SCTLR.C and SCTLR.I bits describe the enabling of the caches, and do not affect the memory attributes generated by an enabled MMU or MPU.

If the MMU or MPU is disabled, the effects of the SCTLR.C and SCTLR.I bits on the memory attributes are described in:

- *Enabling and disabling the MMU* on page B3-5 for the MMU
- *Behavior when the MPU is disabled* on page B4-5 for the MPU.

B2.2.4 Cache maintenance functionality

ARMv7 redefines the required CP15 cache maintenance operations. The two main features of this change are:

- improved support for multiple levels of cache, including abstracting how many levels of cache are implemented.
- reducing the architecturally-defined set of operations to the minimum set required for operating systems

This section only describes cache maintenance for ARMv7. For details of cache maintenance in previous versions of the ARM architecture see:

- *c7, Cache operations* on page AppxG-38 for ARMv6
- *c7, Cache operations* on page AppxH-49 for the ARMv4 and ARMv5 architectures.

Terms used in describing cache operations on page B2-10 describes the terms used in this section. Then the following subsections describe the ARMv7 cache maintenance functionality:

- *ARMv7 cache maintenance operations* on page B2-13
- *The ARMv7 abstraction of the cache hierarchy* on page B2-15.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- 実装で粒度の細かいキャッシュ許可制御が必要な場合は、それを目的とする制御ビットを補助制御レジスタに実装することができます。例えば、特定のレベルでキャッシュを稼動または非稼動にする制御ビットを、実装で定義することができます。補助制御レジスタの詳細については、次のセクションを参照して下さい。
  - *c1、実装定義の補助制御レジスタ (ACTLR)* P. B3-103（VMSA 実装の場合）
  - *c1、実装定義の補助制御レジスタ (ACTLR)* P. B4-50（PMSA 実装の場合）

注

ARMv6 では、SCTLR I、C、W の各ビットによって、レベル 1 命令キャッシュ（実装されている場合）、レベル 1 データまたは統一キャッシュ、およびライトバッファについて個別に稼動が提供されます。詳細については、*c1、システム制御レジスタ (SCTLR)* P. 付録 G-34 を参照して下さい。

キャッシュが非稼動の場合、次の規則が適用されます。

- キャッシュに保持されている位置がアクセスされた場合、キャッシュヒットが発生するかどうかは、実装定義です。
- キャッシュに保持されていない位置は、メモリアクセスの結果としてキャッシュに移動されません。

SCTLR.C ビットと SCTLR.I ビットはキャッシュの稼動を示しますが、稼動中の MMU または MPU で生成されるメモリ属性には影響しません。

MMU または MPU が非稼動の場合、SCTLR.C ビットと SCTLR.I ビットのメモリ属性に対する影響については、次のセクションを参照して下さい。

- *MMU の許可と禁止* P. B3-5（MMU の場合）
- *MPU が稼動していない場合の動作* P. B4-5（MPU の場合）

B2.2.4 キャッシュ保守機能

ARMv7 では、必要な CP15 キャッシュ保守操作が再定義されています。この変更の主な特徴は次のとおりです。

- 何レベルのキャッシュが実装されているかの抽象化など、複数のキャッシュレベルのサポートが改良されています。
- アーキテクチャ定義の操作セットが、オペレーティングシステムに必要な最小セットに削減されています。

このセクションでは、ARMv7 のキャッシュ保守についてのみ説明します。以前のバージョンの ARM アーキテクチャにおけるキャッシュ保守の詳細については、次のセクションを参照して下さい。

- *c7、キャッシュ操作* P. 付録 G-38（ARMv6 の場合）
- *c7、キャッシュ操作* P. 付録 H-49 （ARMv4 および ARMv5 アーキテクチャの場合）

このセクションで使用される用語については、*キャッシュ操作の説明で使用される用語* P. B2-10 で説明されています。また、ARMv7 のキャッシュ保守機能については次のサブセクションで説明されています。

- *ARMv7 のキャッシュ保守操作* P. B2-13
- *ARMv7 のキャッシュ階層の抽象化* P. B2-15

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Terms used in describing cache operations

This section describes particular terms used in the descriptions of cache maintenance operations.

Cache maintenance operations are defined to act on particular memory locations. Operations can be defined:

- by the address of the memory location to be maintained, referred to as by MVA
- by a mechanism that describes the location in the hardware of the cache, referred to as by set/way.

In addition, the instruction cache invalidate operation has an option that invalidates all entries in the instruction caches.

The following subsections define the terms used to describe the cache operations:

- *Operations by MVA*
- *Operations by set/way*
- *Clean, Invalidate, and Clean and Invalidate* on page B2-11.

Operations by MVA

For cache operations by MVA, these terms relate to memory addressing, and in particular the relation between:

- *Modified Virtual Address* (MVA)
- *Virtual Address* (VA)
- *Physical Address* (PA).

The term *Modified Virtual Address* relates to the *Fast Context Switch Extension* (FCSE) mechanism, described in Appendix E *Fast Context Switch Extension (FCSE)*. Use of the FCSE is deprecated in ARMv6 and the FCSE is optional in ARMv7. When the FCSE is absent or disabled, the MVA and VA have the same value. However the term MVA is used throughout this section, and elsewhere in this manual, for cache and TLB operations. This is consistent with previous issues of the *ARM Architecture Reference Manual*.

Virtual addresses only exist in systems with a MMU. When no MMU is implemented or the MMU is disabled, the MVA and VA are identical to the PA.

In the cache operations, any operation described as operating by MVA includes as part of any required MVA to PA translation:

- the current system *Application Space IDentifier* (ASID)
- the current security state, if the Security Extensions are implemented.

Operations by set/way

Cache maintenance operations by set/way refer to the particular structures in a cache. Three parameters describe the location in a cache hierarchy that an operation works on. These parameters are:

Level The cache *level* of the hierarchy. The number of levels of cache is IMPLEMENTATION DEFINED, and can be determined from the Cache Level ID Register, see:

- *c0, Cache Level ID Register (CLIDR)* on page B3-92 for a VMSA implementation
- *c0, Cache Level ID Register (CLIDR)* on page B4-41 for a PMSA implementation.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

キャッシュ操作の説明で使用される用語

このセクションでは、キャッシュ保守操作の説明で使用される特殊な用語について説明します。

キャッシュ保守操作は、特定のメモリ位置に作用するように定義されます。操作は次の方法で定義できます。

- 保守対象のメモリ位置のアドレスによる。MVA による、と呼びます。
- キャッシュ位置を示すハードウェア機構による。セット / ウェイによる、と呼びます。

また、命令キャッシュの無効化操作には、命令キャッシュ内のすべてのエントリを無効化するオプションが含まれています。

キャッシュ操作の説明に使用される用語は、次のサブセクションで定義します。

- *MVA による操作*
- *セット / ウェイによる操作*
- *クリーニング、無効化、およびクリーニングと無効化* P. B2-11

MVA による操作

MVA によるキャッシュ操作の場合は、これらの用語がメモリアドレスの指定、特に相互関係を表します。

- *修飾仮想アドレス*(MVA)
- *仮想アドレス*(VA)
- *物理アドレス*(PA)

*修飾仮想アドレス*という用語は、付録 E *「高速コンテキストスイッチ拡張機能 (FCSE)」*で説明されている*高速コンテキストスイッチ拡張機能*(FCSE) に関連します。FCSE の使用は ARMv6 では推奨されません。ARMv7 では FCSE はオプションです。FCSE が存在しないか、非稼動の場合は、MVA と VA の値は同じです。ただし、MVA という用語は、このセクション全体および本書の他の部分で、キャッシュ操作や TLB 操作に関する説明で使用されます。このことは、旧版の『*ARM アーキテクチャ リファレンスマニュアル*』から一貫しています。

仮想アドレスは、MMU が実装されているシステムにのみ存在します。MMU が存在しないか、非稼動の場合は、MVA と VA は PA と同じ値です。

キャッシュ操作で MVA による操作として記述されている操作には、以下の要素が、必要な MVA/PA 変換の一部として含まれます。

- 現在のシステムの*アプリケーション空間識別子*(ASID)
- セキュリティ拡張機能が実装されている場合、現在のセキュリティ状態

セット / ウェイによる操作

セット / ウェイによるキャッシュ保守操作は、キャッシュ内の特定の構造を参照します。3 つのパラメータによって、操作対象となるキャッシュ階層内の位置が示されます。これらのパラメータには、次のものがあります。

レベル 階層のキャッシュ*レベル*。キャッシュレベル数は実装定義で、キャッシュレベル ID レジスタから特定できます。次のセクションを参照して下さい。

- *c0、キャッシュレベル ID レジスタ (CLIDR)* P. B3-92（VMSA 実装の場合）
- *c0、キャッシュレベル ID レジスタ (CLIDR)* P. B4-41 （PMSA 実装の場合）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In the ARM architecture, the lower numbered levels are those closest to the processor, see *Memory hierarchy* on page A3-52.

Set Each level of a cache is split up into a number of *sets*. Each set is a set of locations in a cache level that an address can be assigned to. Usually, the set number is an IMPLEMENTATION DEFINED function of an address.

In the ARM architecture, sets are numbered from 0.

Way The *Associativity* of a cache defines the number of locations in a set that an address can be assigned to. The *way* number specifies a location in a set. In the ARM architecture, ways are numbered from 0.

Cache maintenance operations that work by set/way use the level, set and way values to determine the location acted on by the operation. The address in memory that corresponds to this cache location is determined by the cache.

Note

Because the allocation of a memory address to a cache location is entirely IMPLEMENTATION DEFINED, ARM expects that most portable code will use only the set/way operations as single steps in a routine to perform maintenance on the entire cache.

Clean, Invalidate, and Clean and Invalidate

Caches introduce coherency problems in two possible directions:

1. An update to a memory location by a processor that accesses a cache might not be visible to other observers that can access memory. This can occur because new updates are still in the cache and are not visible yet to the other observers that do not access that cache.
2. Updates to memory locations by other observers that can access memory might not be visible to a processor that accesses a cache. This can occur when the cache contains an old, or *stale*, copy of the memory location that has been updated.

The *Clean* and *Invalidate* operations address these two issues. The definitions of these operations are:

Clean A cache clean operation ensures that updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the operation is performed. Once the Clean has completed, the new memory values are guaranteed to be visible to the point to which the operation is performed, for example to the point of unification.

The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by a processor that controls that cache.

Invalidate A cache invalidate operation ensures that updates made visible by observers that access memory at the point to which the invalidate is defined are made visible to an observer that controls the cache. This might result in the loss of updates to the locations affected by the invalidate operation that have been written by observers that access the cache.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARM アーキテクチャでは、番号が小さいレベルほどプロセッサの近くに位置します。*メモリの階層* P. A3-52 を参照して下さい。

セット 各キャッシュレベルは、複数の *セット*に分割されます。各セットは、アドレスを割り当てることが可能なキャッシュレベル内の一連の位置です。通常は、セット番号はアドレスの実装定義の関数です。

ARM アーキテクチャでは、セットに 0 から番号が付けられます。

ウェイ キャッシュの *アソシエティビティ*によって、アドレスを割り当てることが可能なセット内の位置の数が定義されます。*ウェイ*番号は、セット内の特定の位置を示します。ARM アーキテクチャでは、ウェイに 0 から番号が付けられます。

セット / ウェイによるキャッシュ保守操作では、レベル、セット、ウェイの値を使用して操作対象の位置が決定されます。このキャッシュ位置に対応するメモリ内のアドレスは、キャッシュによって決定されます。

注

キャッシュ位置に対するメモリアドレスの割り当ては完全に実装定義のため、ARM では、ほとんどの移植可能コードで、キャッシュ全体を保守するルーチン内の手順の 1 つとして、セット / ウェイ操作のみが使用されることを前提としています。

クリーニング、無効化、およびクリーニングと無効化

キャッシュでは、次の 2 つの方向でコヒーレンシの問題が発生します。

1. キャッシュにアクセスするプロセッサによるメモリ位置の更新が、メモリにアクセス可能な他のオブザーバから可視でない可能性があります。この現象は、新しい更新がまだキャッシュ内にあり、そのキャッシュにアクセスしていない他のオブザーバから可視でないことから発生する可能性があります。
2. メモリにアクセス可能な他のオブザーバによるメモリ位置の更新が、キャッシュにアクセスするプロセッサから可視でない可能性があります。この現象は、更新されたメモリ位置の古い、つまり*失効した*コピーがキャッシュに含まれている場合に発生する可能性があります。

*クリーニング*操作と*無効化*操作によって、この 2 つの問題が解決されます。これらの操作は、次のように定義されます。

クリーニング キャッシュのクリーニング操作により、キャッシュを制御するオブザーバによって行われた更新が、操作が実行された時点でメモリにアクセス可能な他のオブザーバから可視になることが保証されます。クリーニングが完了すると、新しいメモリの値が、操作が実行された時点、例えば統一の時点まで可視になることが保証されます。

キャッシュのキャッシュエントリをクリーニングするとき、そのエントリにそのキャッシュを制御しているプロセッサから書き込まれたメモリ位置が含まれている場合にのみ、他のオブザーバによって書き込まれたメモリを上書きすることができます。

無効化 キャッシュの無効化操作では、無効化が定義された時点でメモリにアクセスするオブザーバにによって可視になった更新が、キャッシュを制御するオブザーバから可視になることが保証されます。これによって、無効化操作の影響を受けた位置に対して、キャッシュにアクセスするオブザーバから書き込まれた更新が失われる可能性があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If the address of an entry on which the invalidate operates does not have a Normal Cacheable attribute, or if the cache is disabled, then an invalidate operation also ensures that this address is not present in the cache.

Note

Entries for addresses with a Normal Cacheable attribute can be allocated to an enabled cache at any time, and so the cache invalidate operation cannot ensure that the address is not present in the cache.

Clean and Invalidate

A cache *clean and invalidate* operation behaves as the execution of a clean operation followed immediately by an invalidate operation. Both operations are performed to the same location.

The points to which a cache maintenance operation can be defined differ depending on whether the operation is by MVA or by set/way:

- For set/way operations, and for *All* (entire cache) operations, the point is defined to be to the next level of caching.
- For MVA operations, two conceptual points are defined:

Point of coherency (POC)

For a particular MVA, the POC is the point at which all agents that can access memory are guaranteed to see the same copy of a memory location. In many cases, this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the POC that have no effect on the coherence between memory system agents.

Point of unification (POU)

The PoU for a processor is the point by which the instruction and data caches and the translation table walks of that processor are guaranteed to see the same copy of a memory location. In many cases, the point of unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged.

The PoU for an Inner Shareable shareability domain is the point by which the instruction and data caches and the translation table walks of all the processors in that Inner Shareable shareability domain are guaranteed to see the same copy of a memory location.

Defining this point permits self-modifying code to ensure future instruction fetches are associated with the modified version of the code by using the standard correctness policy of:

1. clean data cache entry by address
2. invalidate instruction cache entry by address.

The PoU also enables a uniprocessor system which does not implement the Multiprocessing Extensions to use the clean data cache entry operation to ensure that all writes to the translation tables are visible to the translation table walk hardware.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

無効化操作対象のエントリのアドレスにノーマルキャッシュ可能属性が付与されていない場合、またはキャッシュが非稼動の場合は、無効化操作によって、そのアドレスがキャッシュ内に存在しないことも保証されます。

注

ノーマルキャッシュ可能属性が付与されたアドレスのエントリは、いつでも許可されたキャッシュに割り当てることができるため、キャッシュ無効化操作によって、そのアドレスがキャッシュに存在しないことは保証できません。

クリーニングと無効化

キャッシュの*クリーニングと無効化*操作は、クリーニングの直後に無効化が実行される操作です。両方の操作が同じ位置に対して実行されます。

キャッシュ保守操作が定義可能な時点は、操作が MVA とセット / ウェイのどちらで行われるかによって異なります。

- セット / ウェイ操作と*全体*（キャッシュ全体）操作の場合は、次のレベルのキャッシュに対して時点が定義されます。
- MVA 操作の場合は、次の 2 つの概念的な時点が定義されます。

コヒーレンシ時点 (POC)

特定の MVA の POC は、メモリにアクセス可能なすべてのエージェントが、同じメモリ位置のコピーを認識することが保証される時点です。多くの場合、これは事実上のメイン システムメモリですが、アーキテクチャ上は、メモリシステム エージェント間のコヒーレンシに影響を与えない、POC 以後のキャッシュの実装は禁止されていません。

統一時点 (POU)

プロセッサの POU は、命令キャッシュ、データキャッシュ、およびそのプロセッサの変換テーブルウォークが、同じメモリ位置のコピーの認識を保証される時点です。多くの場合、統一時点は、命令キャッシュ、データキャッシュ、および変換テーブルウォークがマージされるユニプロセッサ メモリシステム内の時点です。

内部共有可能な共有可ドメインの POU は、命令キャッシュ、データキャッシュ、およびその内部共有可能な共有可ドメイン内のすべてのプロセッサの変換テーブルウォークが、同じメモリ位置のコピーの認識を保証される時点です。

この時点の定義によって、自己修正コードでは次のような標準的な正確性ポリシーを使用し、将来の命令フェッチが変更されたバージョンのコードに関連付けられることを保証できます。

1. アドレスによるデータキャッシュ エントリのクリーニング
2. アドレスによる命令キャッシュエントリの無効化

POU を使用すれば、マルチプロセッシング拡張機能が実装されていないユニプロセッサシステムで、データキャッシュ エントリのクリーンニング操作を使用して、変換テーブルへのすべての書き込みが変換テーブルウォーク ハードウェアから可視であることを保証することもできます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Three field definitions in the Cache Level ID Register relate to these conceptual points:

Level of Coherence

The level of coherence field defines the first level of cache that does not have to be cleaned or invalidated when cleaning or invalidating to the point of coherency. The value in the register is one less than the cache level, so a value of 0 indicates level 1 cache. For example, if the level of coherence field contains the value 3:

- level 4 cache is the first level that does not have to be cleaned or invalidated
- therefore, a clean to the point of coherency operation requires the level 1, level 2 and level 3 caches to be cleaned.

The specified level of coherence can be a level that is not implemented, indicating that all implemented caches are before the point of coherency.

Level of Unification Uniprocessor

The Level of Unification Uniprocessor field defines the first level of cache that does not have to be cleaned or invalidated when cleaning or invalidating to the point of unification for the processor. As with the Level of Coherence, the value in the register is one less than the cache level, so a value of 0 indicates Level 1 cache.

The specified Level of Unification Uniprocessor can be a level that is not implemented, indicating that all implemented caches are before the point of unification.

Level of Unification Inner Shareable

The Level of Unification Inner Shareable field defines the first level of cache that does not have to be cleaned or invalidated when cleaning or invalidating to the point of unification for the Inner Shareable shareability domain. As with the Level of Coherence, the value in the register is one less than the cache level, that means a value of 0 indicates Level 1 cache.

The specified Level of Unification Inner Shareable can be a level that is not implemented, indicating that all implemented caches are before the point of unification.

The Level of Unification Inner Shareable field is RAZ in implementations that do not implement the Multiprocessing Extensions.

For more information, see:

— *c0, Cache Level ID Register (CLIDR)* on page B3-92 for a VMSA implementation

— *c0, Cache Level ID Register (CLIDR)* on page B4-41 for a PMSA implementation.

ARMv7 cache maintenance operations

Cache maintenance operations are performed using accesses to CP15 c7. The operations are described in:

- *CP15 c7, Cache and branch predictor maintenance functions* on page B3-126, for a VMSA implementation
- *CP15 c7, Cache and branch predictor maintenance functions* on page B4-68, for a PMSA implementation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

キャッシュレベル ID レジスタ内の次の 3 つのフィールド定義が、これらの概念上の時点に関係します。

コヒーレンシレベル

コヒーレンシレベル フィールドでは、コヒーレンシ時点までのクリーニングまたは無効化時にクリーニングまたは無効化する必要のない、最初のキャッシュレベルが定義されます。このレジスタの値はキャッシュレベルより 1 小さいため、0 の値がレベル 1 キャッシュを示します。例えば、コヒーレンシレベル フィールドの値が 3 の場合は、次のようになります。

- レベル 4 キャッシュが、クリーニングまたは無効化する必要のない最初のレベルです。
- したがって、コヒーレンシ時点までのクリーニング操作で、レベル 1、レベル 2、レベル 3 のキャッシュをクリーニングする必要があります。

指定するコヒーレンシレベルは、実装されたすべてのキャッシュがコヒーレンシ時点前であることを表す、実装されていないレベルにすることができます。

ユニプロセッサ統一レベル

ユニプロセッサ統一レベル フィールドでは、プロセッサの統一時点までのクリーニングまたは無効化時にクリーニングまたは無効化する必要のない、最初のキャッシュレベルが定義されます。コヒーレンシレベルと同様に、このレジスタ内の値はキャッシュレベルより 1 小さいため、0 の値がレベル 1 キャッシュを表します。

指定するユニプロセッサ統一レベルは、実装されたすべてのキャッシュが統一時点前であることを表す、実装されていないレベルにすることができます。

内部共有可能統一レベル

内部共有可能統一レベルフィールドでは、内部共有可能な共有可ドメインの統一時点までクリーニングまたは無効化を行う時にクリーニングまたは無効化する必要のない、最初のキャッシュレベルが定義されます。コヒーレンシレベルと同様に、このレジスタ内の値はキャッシュレベルより 1 小さいため、0 の値がレベル 1 キャッシュを表します。

指定する内部共有可能統一レベルは、実装されたすべてのキャッシュが統一時点前であることを表す、実装されていないレベルにすることができます。

マルチプロセッシング拡張機能を含まない実装では、内部共有可能統一レベルフィールドは RAZ です。

詳細については、次の各セクションを参照して下さい。

– *c0、キャッシュレベル ID レジスタ (CLIDR)* P. B3-92（VMSA 実装の場合）

– *c0、キャッシュレベル ID レジスタ (CLIDR)* P. B4-41 （PMSA 実装の場合）

ARMv7 のキャッシュ保守操作

キャッシュ保守操作は、CP15 c7 にアクセスすることによって実行されます。この操作については、次の各セクションを参照して下さい。

- *CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126（VMSA 実装の場合）
- *CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68（PMSA 実装の場合）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

This operations required by ARMv7 are:

Data cache and unified cache line operations

Any of these operations can be applied to

- any data cache
- any unified cache.

The supported operations are:

Invalidate by MVA

Performs an invalidate of a data or unified cache line based on the address it contains.

Invalidate by set/way

Performs an invalidate of a data or unified cache line based on its location in the cache hierarchy.

Clean by MVA

Performs a clean of a data or unified cache line based on the address it contains.

Clean by set/way

Performs a clean of a data or unified cache line based on its location in the cache hierarchy.

Clean and Invalidate by MVA

Performs a clean and invalidate of a data or unified cache line based on the address it contains.

Clean and Invalidate by set/way

Performs a clean and invalidate of a data or unified cache line based on its location in the cache hierarchy.

Instruction cache operations

Invalidate by MVA

Performs an invalidate of an instruction cache line based on the address it contains.

Invalidate All

Performs an invalidate of the entire instruction cache or caches, and of all Branch Prediction caches.

Note

Other cache maintenance operations specified in ARMv6 are not supported in ARMv7. Their associated encodings in CP15 c7 are UNPREDICTABLE.

An ARMv7 implementation can add additional IMPLEMENTATION DEFINED cache maintenance functionality using CP15 c15 operations, if this is required.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ARMv7 で必要な操作は次のとおりです。

データキャッシュおよび統一キャッシュのラインの操作

これらの操作は、次のものに適用できます。

- 任意のデータキャッシュ
- 任意の統一キャッシュ

サポートされている操作は次のとおりです。

MVA による無効化

含まれているアドレスに基づいて、データまたは統一キャッシュラインの無効化を実行します。

セット / ウェイによる無効化

キャッシュ階層内の位置に基づいて、データまたは統一キャッシュラインの無効化を実行します。

MVA によるクリーニング

含まれているアドレスに基づいて、データまたは統一キャッシュラインのクリーニングを実行します。

セット / ウェイによるクリーニング

キャッシュ階層内の位置に基づいて、データまたは統一キャッシュラインのクリーニングを実行します。

MVA によるクリーニングと無効化

含まれているアドレスに基づいて、データまたは統一キャッシュラインのクリーニングと無効化を実行します。

セット / ウェイによるクリーニングと無効化

キャッシュ階層内の位置に基づいて、データまたは統一キャッシュラインのクリーニングと無効化を実行します。

命令キャッシュ操作

MVA による無効化

含まれているアドレスに基づいて、命令キャッシュラインの無効化を実行します。

全体の無効化

命令キャッシュ全体と分岐予測キャッシュ全体の無効化を実行します。

注

ARMv6 で指定されたその他のキャッシュ保守操作は、ARMv7 ではサポートされていません。これらに関連した CP15 c7 内のエンコードは予測不能です。

ARMv7 実装では、必要に応じて、CP15 c15 操作を使用した新しい実装定義のキャッシュ保守機能を追加することができます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The ARMv7 specification of the cache maintenance operation describe what they are guaranteed to do in a system. It does not limit other behaviors that might occur, provided they are consistent with the requirements for cache behavior described in *Cache behavior* on page B2-5.

This means that as a side-effect of a cache maintenance operation:

- any location in the cache might be cleaned
- any unlocked location in the cache might be cleaned and invalidated.

Note

ARM recommends that, for best performance, such side-effects are kept to a minimum. In particular, when the Security Extensions are implemented ARM strongly recommends that the side-effects of operations performed in Non-secure state do not have a significant performance impact on execution in Secure state.

Effect of the Security Extensions on the cache maintenance operations

When the Security Extensions are implemented, each security state has its own physical address space. For details of how this affects the cache maintenance operations see *The effect of the Security Extensions on the cache operations* on page B3-27.

The ARMv7 abstraction of the cache hierarchy

The following subsections describe the ARMv7 abstraction of the cache hierarchy:

- *Cache hierarchy abstraction for address-based operations*
- *Cache hierarchy abstraction for set/way-based operations* on page B2-16.

Example code for cache maintenance operations on page B2-16 gives an example of cache maintenance code, that can be adapted for other cache operations, and *Boundary conditions for cache maintenance operations* on page B2-17 gives more information about the cache operations.

Cache hierarchy abstraction for address-based operations

The addressed-based cache operations are described as operating by MVA. Each of these operations is always qualified as being one of:

- performed to the point of coherency
- performed to the point of unification.

See *Terms used in describing cache operations* on page B2-10 for definitions of point of coherency and point of unification, and more information about possible meanings of MVA.

This means that the full list of possible address-based cache operations is:

- Invalidate data cache or unified cache line by MVA to the point of coherency
- Clean data cache or unified cache line by MVA to the point of coherency
- Clean data cache or unified cache line by MVA to the point of unification
- Clean and invalidate data cache or unified cache line by MVA to the point of coherency
- Invalidate instruction cache line by MVA to the point of unification.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

キャッシュ保守操作の ARMv7 仕様には、それらの操作がシステム内でどのような効力を持つことが保証されるかが記載されています。その他の動作が発生することは、*キャッシュの動作* P. B2-5 で説明されているキャッシュの動作の要件と整合している限りは制限されません。

これは、キャッシュ保守操作の副作用として、次の可能性があることを意味します。

- キャッシュ内のいずれかの位置がクリーニングされる。
- キャッシュ内のロックされていないいずれかの位置がクリーニングおよび無効化される。

注

パフォーマンスを最大化するため、このような副作用は最小限に抑えることをお勧めします。特に、セキュリティ拡張機能が実装されている場合は、非セキュア状態で実行される操作の副作用が、セキュア状態での実行性能に悪影響を及ぼさないようにすることを強くお勧めします。

キャッシュ保守操作に対するセキュリティ拡張機能の影響

セキュリティ拡張機能が実装されている場合は、セキュリティ状態ごとに物理的なアドレス空間が割り当てられます。このことがキャッシュ保守操作に与える影響の詳細については、*キャッシュ操作に対するセキュリティ拡張機能の作用* P. B3-27 を参照して下さい。

ARMv7 のキャッシュ階層の抽象化

次のサブセクションでは、ARMv7 のキャッシュ階層の抽象化について説明します。

- *アドレスベースの操作でのキャッシュ階層抽象化*
- *セット／ウェイベースの操作に関するキャッシュ階層の抽象化* P. B2-16

キャッシュ保守操作のサンプルコード P. B2-16 では、他のキャッシュ操作に適用可能なキャッシュ保守コードの例を示します。また、*キャッシュ保守操作の境界条件* P. B2-17 では、キャッシュ操作について詳しく説明します。

アドレスベースの操作でのキャッシュ階層抽象化

アドレスベースのキャッシュ操作は、MVA による操作として記述されます。このような操作のそれぞれは、必ず次のどちらかに修飾されます。

- コヒーレンシ時点まで実行される。
- 統一時点まで実行される。

コヒーレンシ時点および統一時点の定義と、MVA について可能な各種の意味については、*キャッシュ操作の説明で使用される用語* P. B2-10 を参照して下さい。

この結果、使用可能なアドレスベースのキャッシュ操作は次に示すもののみです。

- コヒーレンシ時点までの、MVA によるデータまたは統一キャッシュラインの無効化
- コヒーレンシ時点までの、MVA によるデータまたは統一キャッシュラインのクリーニング
- 統一時点までの、MVA によるデータまたは統一キャッシュラインのクリーニング
- コヒーレンシ時点までの、MVA によるデータまたは統一キャッシュラインのクリーニングと無効化
- 統一時点までの、MVA による命令キャッシュラインの無効化

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The Cache Type Register holds minimum line length values for:

- the instruction caches
- the data and unified caches.

These values enable a range of addresses to be invalidated in an efficient manner. For details of the register see:

- *c0, Cache Type Register (CTR)* on page B3-83 for a VMSA implementation
- *c0, Cache Type Register (CTR)* on page B4-34 for a PMSA implementation.

For details of the CP15 c7 encodings for all cache maintenance operations see:

- *CP15 c7, Cache and branch predictor maintenance functions* on page B3-126 for a VMSA implementation
- *CP15 c7, Cache and branch predictor maintenance functions* on page B4-68 for a PMSA implementation.

Cache hierarchy abstraction for set/way-based operations

The set/way-based cache maintenance operations are:

- Invalidate data cache or unified cache line by set/way
- Clean data cache or unified cache line by set/way
- Clean and invalidate data cache or unified cache line by set/way

The CP15 c7 encodings of these operations include a field that must be used to specify the cache level for the operation:

- a clean operation cleans from the level of cache specified through to at least the next level of cache, moving further from the processor
- an invalidate operation invalidates only at the level specified.

In addition to these set/way operations, a cache operation is provided for instruction cache maintenance, to Invalidate all instruction cache lines to the point of unification.

For details of the CP15 c7 encodings for all cache maintenance operations see:

- *CP15 c7, Cache and branch predictor maintenance functions* on page B3-126 for a VMSA implementation
- *CP15 c7, Cache and branch predictor maintenance functions* on page B4-68 for a PMSA implementation.

Example code for cache maintenance operations

This code sequence illustrates a generic mechanism for cleaning the entire data or unified cache to the point of coherency:

```
        MRC p15, 1, R0, c0, c0, 1     ; Read CLIDR
        ANDS R3, R0, #&7000000
        MOV R3, R3, LSR #23           ; Cache level value (naturally aligned)
        BEQ Finished
        MOV R10, #0
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

 抄訳

キャッシュタイプ レジスタには、次の最小ライン長の値が保持されます。

- 命令キャッシュ
- データキャッシュと統一キャッシュ

これらの値によって、アドレスの範囲を効率的に無効化することができます。レジスタの詳細については、次のセクションを参照して下さい。

- *c0、キャッシュタイプ レジスタ (CTR)* P. B3-83（VMSA 実装の場合）
- *c0、キャッシュタイプ レジスタ (CTR)* P. B4-34 （PMSA 実装の場合）

すべてのキャッシュ保守操作についての CP15 c7 エンコードの詳細については、次のセクションを参照して下さい。

- *CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126（VMSA 実装の場合）
- *CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68 （PMSA 実装の場合）

セット／ウェイベースの操作に関するキャッシュ階層の抽象化

セット / ウェイベースのキャッシュ保守操作は次のとおりです。

- セット / ウェイによるデータまたは統一キャッシュラインの無効化
- セット / ウェイによるデータまたは統一キャッシュラインのクリーニング
- セット / ウェイによるデータまたは統一キャッシュラインのクリーニングと無効化

これらの操作の CP15 c7 エンコードには、操作のキャッシュレベルを指定するために使用する必要のあるフィールドが含まれています。

- クリーニング操作では、指定されたキャッシュレベルから少なくとも次のキャッシュレベルまで、プロセッサから遠ざかる方向にクリーニングされます。
- 無効化操作は、指定されたレベルでのみ実行されます。

命令キャッシュの保守では、これらのセット / ウェイ操作に加えて、統一時点までのすべての命令キャッシュラインを無効化するためのキャッシュ操作が提供されます。

すべてのキャッシュ保守操作についての CP15 c7 エンコードの詳細については、次のセクションを参照して下さい。

- *CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126（VMSA 実装の場合）
- *CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68 （PMSA 実装の場合）

キャッシュ保守操作のサンプルコード

このコードシーケンスは、コヒーレンシ時点までデータキャッシュまたは統一キャッシュ全体をクリーニングする、汎用の機構を示しています。

```
        MRC p15, 1, R0, c0, c0, 1     ; Read CLIDR
        ANDS R3, R0, #&7000000
        MOV R3, R3, LSR #23           ; Cache level value (naturally aligned)
        BEQ Finished
        MOV R10, #0
Loop1   ADD R2, R10, R10, LSR #1      ; Work out 3xcachelevel
        MOV R1, R0, LSR R2            ; bottom 3 bits are the Cache type for this level
        AND R1, R1, #7                ; get those 3 bits alone
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
Loop1   ADD R2, R10, R10, LSR #1      ; Work out 3xcachelevel
        MOV R1, R0, LSR R2            ; bottom 3 bits are the Cache type for this level
        AND R1, R1, #7                ; get those 3 bits alone
        CMP R1, #2
        BLT Skip                      ; no cache or only instruction cache at this level
        MCR p15, 2, R10, c0, c0, 0    ; write the Cache Size selection register
        ISB                           ; ISB to sync the change to the CacheSizeID reg
        MRC p15, 1, R1, c0, c0, 0     ; reads current Cache Size ID register
        AND R2, R1, #&7               ; extract the line length field
        ADD R2, R2, #4                ; add 4 for the line length offset (log2 16 bytes)
        LDR R4, =0x3FF
        ANDS R4, R4, R1, LSR #3       ; R4 is the max number on the way size (right aligned)
        CLZ R5, R4                    ; R5 is the bit position of the way size increment
        LDR R7, =0x00007FFF
        ANDS R7, R7, R1, LSR #13      ; R7 is the max number of the index size (right aligned)
Loop2   MOV R9, R4                    ; R9 working copy of the max way size (right aligned)
Loop3   ORR R11, R10, R9, LSL R5      ; factor in the way number and cache number into R11
        ORR R11, R11, R7, LSL R2      ; factor in the index number
        MCR p15, 0, R11, c7, c10, 2   ; clean by set/way
        SUBS R9, R9, #1               ; decrement the way number
        BGE Loop3
        SUBS R7, R7, #1               ; decrement the index
        BGE Loop2
Skip    ADD R10, R10, #2              ; increment the cache number
        CMP R3, R10
        BGT Loop1
Finished
```

Similar approaches can be used for all cache maintenance operations.

Boundary conditions for cache maintenance operations

Cache maintenance operations operate on the caches when the caches are enabled or when they are disabled.

For the address-based cache maintenance operations, the operations operate on the caches regardless of the memory type and cacheability attributes marked for the memory address in the VMSA translation table entries or in the PMSA section attributes. This means that the cache operations take no account of:

- whether the address accessed:
  - — is Strongly-ordered, Device or Normal memory
  - — has a Cacheable attribute or the Non-cacheable attribute
- the domain control of the address accessed
- the access permissions for the address accessed.

Therefore, software can:

- ensure there are no more allocations to the caches of a range of addresses because of prefetching effects or interrupts
- at the same time, continue to perform cache maintenance operations on these addresses.

In a VMSA implementation, some cache maintenance operations can generate an MMU fault, see *MMU faults* on page B3-40.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
        CMP R1, #2
        BLT Skip                      ; no cache or only instruction cache at this level
        MCR p15, 2, R10, c0, c0, 0    ; write the Cache Size selection register
        ISB                           ; ISB to sync the change to the CacheSizeID reg
        MRC p15, 1, R1, c0, c0, 0     ; reads current Cache Size ID register
        AND R2, R1, #&7               ; extract the line length field
        ADD R2, R2, #4                ; add 4 for the line length offset (log2 16 bytes)
        LDR R4, =0x3FF
        ANDS R4, R4, R1, LSR #3       ; R4 is the max number on the way size (right aligned)
        CLZ R5, R4                    ; R5 is the bit position of the way size increment
        LDR R7, =0x00007FFF
        ANDS R7, R7, R1, LSR #13      ; R7 is the max number of the index size (right aligned)
Loop2   MOV R9, R4                    ; R9 working copy of the max way size (right aligned)
Loop3   ORR R11, R10, R9, LSL R5      ; factor in the way number and cache number into R11
        ORR R11, R11, R7, LSL R2      ; factor in the index number
        MCR p15, 0, R11, c7, c10, 2   ; clean by set/way
        SUBS R9, R9, #1               ; decrement the way number
        BGE Loop3
        SUBS R7, R7, #1               ; decrement the index
        BGE Loop2
Skip    ADD R10, R10, #2              ; increment the cache number
        CMP R3, R10
        BGT Loop1
Finished
```

すべてのキャッシュ保守操作に対して、同様の手法が使用できます。

キャッシュ保守操作の境界条件

キャッシュ保守操作は、キャッシュが稼動中の場合、またはキャッシュが非稼動の場合に、キャッシュに対して実行されます。

アドレスベースのキャッシュ保守操作は、VMSA 変換テーブルエントリまたは PMSA セクション属性のメモリアドレスにマークされたメモリタイプとキャッシュ可能の属性に関係なく、キャッシュに対して実行されます。つまり、キャッシュ操作では、次のことは考慮されません。

- アクセスされるアドレスの属性
  - ストロングリオーダ、デバイス、ノーマルのどのメモリか。
  - キャッシュ可能属性と、キャッシュ不可属性のどちらであるか。
- アクセスされるアドレスのドメイン制御
- アクセスされるアドレスのアクセス許可

このため、ソフトウェアは次の操作を実行できます。

- プリフェッチの影響または割り込みによって、アドレスの範囲のキャッシュに対する割り当てがこれ以上増えないことを保証する。
- 同時に、これらのアドレスに対するキャッシュ保守操作の実行を継続する。

VMSA 実装では、一部のキャッシュ保守操作で MMU フォールトが発生する可能性があります。*MMU フォールト* P. B3-40 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.2.5 The interaction of cache lockdown with cache maintenance

The interaction of cache lockdown and cache maintenance operations is IMPLEMENTATION DEFINED. However, an architecturally-defined cache maintenance operation on a locked cache line must comply with the following general rules:

- The effect of these operations on locked cache entries is IMPLEMENTATION DEFINED:
  - — cache clean by set/way
  - — cache invalidate by set/way
  - — cache clean and invalidate by set/way
  - — instruction cache invalidate all.

  However, one of the following approaches must be adopted in all these cases:

  1. If the operation specified an invalidation a locked entry is not invalidated from the cache. If the operation specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
  2. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the fault status code defined for this purpose in CP15 c5, see either:
     - *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48
     - *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18.

  This permits a typical usage model for cache invalidate routines to operate on a large range of addresses by performing the required operation on the entire cache, without having to consider whether any cache entries are locked. The operation performed is either an invalidate, or a clean and invalidate.

- The effect of these operations is IMPLEMENTATION DEFINED:
  - — cache clean by MVA
  - — cache invalidate by MVA
  - — cache clean and invalidate by MVA.

  However, one of the following approaches must be adopted in all these cases:

  1. If the operation specified an invalidation a locked entry is invalidated from the cache. For the clean and invalidate operation, the entry must be cleaned before it is invalidated.
  2. If the operation specified an invalidation a locked entry is not invalidated from the cache. If the operation specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned
  3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the fault status code defined for this purpose in CP15 c5, see either:
     - *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48
     - *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18.

An implementation that uses the abort mechanisms for entries that could be locked must:

- document IMPLEMENTATION DEFINED code sequences that then perform the required operation on entries that are not locked down

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.2.5 キャッシュロックダウンとキャッシュ保守との相互動作

キャッシュロックダウンとキャッシュ保守操作の相互動作は、実装定義です。ただし、ロックされたキャッシュラインに対するアーキテクチャ定義のキャッシュ保守操作は、以下の一般的な規則に従う必要があります。

- ロックされたキャッシュエントリに対するこれらの操作の影響は、実装定義です。
  - – セット / ウェイによるキャッシュのクリーニング
  - – セット / ウェイによるキャッシュの無効化
  - – セット / ウェイによるキャッシュのクリーニングと無効化
  - – 命令キャッシュ全体の無効化

  ただし、次の手法のいずれかを、これらの場合すべてについて採用する必要があります。

  1. 操作で無効化が指定されても、ロックされたエントリはキャッシュから無効化されません。操作でクリーニングが指定されても、ロックされたエントリがクリーニングされるかどうかは実装定義です。
  2. エントリがロックダウンされている場合、またはロックダウンされた可能性がある場合は、この目的のために CP15 c5 に定義されているフォールトステータス コードを使用して、実装定義のデータアボート例外が生成されます。次のセクションを参照して下さい。
     - *VMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B3- 48
     - *PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4- 18

  これによって、キャッシュ無効化ルーチンの一般的な使用モデルでは、キャッシュエントリがロックされているかどうかを気にすることなく、キャッシュ全体に対して必要な操作を実行して、広い範囲のアドレスに対して効果を及ぼすことができます。実行される操作は、無効化、またはクリーニングと無効化です。

- 次の操作の影響は実装定義です。
  - – MVA によるキャッシュのクリーニング
  - – MVA によるキャッシュの無効化
  - – MVA によるキャッシュのクリーンニングと無効化

  ただし、次の手法のいずれかを、これらの場合すべてについて採用する必要があります。

  1. 操作で無効化が指定された場合は、ロックされたエントリがキャッシュから無効化されます。クリーニングと無効化操作の場合は、エントリを無効化する前にクリーニングする必要があります。
  2. 操作で無効化が指定されても、ロックされたエントリはキャッシュから無効化されません。操作でクリーニングが指定されても、ロックされたエントリがクリーニングされるかどうかは実装定義です。
  3. エントリがロックダウンされている場合、またはロックダウンされた可能性がある場合は、この目的のために CP15 c5 に定義されているフォールトステータス コードを使用して、実装定義のデータアボート例外が生成されます。次のセクションを参照して下さい。
     - *VMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B3- 48
     - *PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4- 18

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- implement one of the other permitted alternatives for the locked entries.

ARM recommends that, where possible, architecturally-defined operations are used in such code sequences. This minimizes the number of customized operations required.

In addition, any implementation that uses aborts for handling cache maintenance operations on entries that might be locked must provide a mechanism that can be used to ensures that no entries are locked in the cache. The reset setting of the cache must be that no cache entries are locked.

On an ARMv7-A implementation, similar rules apply to TLB lockdown, see *The interaction of TLB maintenance operations with TLB lockdown* on page B3-57.

Additional cache functions for the implementation of lockdown

An implementation can add additional cache maintenance functions for the handling of lockdown in the IMPLEMENTATION DEFINED spaces reserved for Cache Lockdown. Examples of possible functions are:

- Operations that unlock all cache entries.
- Operations that preload into specific levels of cache. These operations might be provided for instruction caches, data caches, or both.

An implementation can add other functions as required.

B2.2.6 Branch predictors

Branch predictor hardware typically uses a form of cache to hold branch information. The ARM architecture permits this branch predictor hardware to be visible to the functional behavior of software, and so the branch predictor is not architecturally invisible. This means that under some circumstances software must perform branch predictor maintenance to avoid incorrect execution caused by out of date entries in the branch predictor.

Branch prediction maintenance operations

In some implementations, to ensure correct operation it might be necessary to invalidate branch prediction entries on a change of instruction or instruction address mapping. For more information, see *Branch predictor maintenance operations and the memory order model* on page B2-20.

Two CP15 c7 operations apply to branch prediction hardware, these two functions are:

```
MCR p15, 0, Rt, c7, c5, 6:    Invalidate entire branch predictor array
MCR p15, 0, Rt, c7, c5, 7:    Invalidate MVA from branch predictor array
```

In ARMv7, these functions can perform a NOP if the operation of Branch Prediction hardware is not visible architecturally.

The invalidate entire branch predictor array operation ensures that any location held in the branch predictor has no functional effect on execution.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ロックされている可能性のあるエントリについてアボート機構を使用する実装については、次の作業を行う必要があります。

- ロックダウンされていないエントリに対して必要な操作を実行する実装定義のコードシーケンスを文書化する。
- ロックされたエントリに関する許可された代替手段の 1 つを実装する。

このようなコードシーケンスでは、可能な限りアーキテクチャ定義の操作を使用することをお勧めします。これによって、カスタマイズが必要な操作の数が最小化されます。

さらに、ロックされた可能性のあるエントリに対するキャッシュ保守操作をアボートで処理する実装では、キャッシュ内のどのエントリもロックされていないことを保証するための機構を提供する必要があります。キャッシュのリセット時の設定は、どのキャッシュエントリもロックされていない状態にする必要があります。

ARMv7-A 実装では、同様の規則が TLB ロックダウンに適用されます。*TLB 保守操作と TLB ロックダウンとの相互動作* P. B3-57 を参照して下さい。

ロックダウンの実装のために追加されたキャッシュ関数

キャッシュロックダウン用に予約された実装定義空間でロックダウンを処理するための、新しいキャッシュ保守機能を実装に追加できます。例えば、次のような機能が使用可能です。

- すべてのキャッシュエントリをロック解除する操作
- 特定のキャッシュレベルにプリロードする操作。この操作は、命令キャッシュとデータキャッシュのどちらか、またはその両方に対して提供することができます。

必要に応じて、他の機能を実装に追加できます。

B2.2.6 分岐予測器

分岐予測器ハードウェアは一般に、何らかの形式のキャッシュに分岐情報を保持します。ARM アーキテクチャでは、分岐予測器ハードウェアがソフトウェアの機能的動作から可視であることが許されるため、分岐予測器はアーキテクチャ上不可視ではありません。これは、一部の状況では、ソフトウェアで分岐予測器の保守を実行し、分岐予測器内の期限切れのエントリによる不正な実行を防ぐ必要があることを意味します。

分岐予測の保守操作

一部の実装では、正しい操作を保証するために、命令または命令アドレスマッピングが変化したときに分岐予測エントリを無効化することが必要な可能性があります。詳細については、*分岐予測器の保守操作とメモリオーダ モデル* P. B2-20 を参照して下さい。

次の 2 つの CP15 c7 操作が分岐予測ハードウェアに適用されます。

```
MCR p15, 0, Rt, c7, c5, 6:    Invalidate entire branch predictor array
MCR p15, 0, Rt, c7, c5, 7:    Invalidate MVA from branch predictor array
```

ARMv7 では、分岐予測ハードウェアの操作がアーキテクチャ上不可視になっている場合は、これらの命令が NOP として実行される可能性があります。

分岐予測器アレイ全体の無効化操作によって、分岐予測器に保持されているすべての位置が、実行に機能的な影響を及ぼさないことが保証されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The invalidate MVA from branch predictor array operation operates on the address of the branch instruction. It includes the current system ASID and the security state when determining which line is affected as part of any required VA to PA translation. Security state checking is performed only if the Security Extensions are implemented. The invalidate by MVA operation can affect other branch predictor entries.

Note

The architecture does not make visible the range of addresses in a branch predictor to which the invalidate operation applies. This means the address used in the invalidate MVA instruction must be the address of the branch to be invalidated.

If the correct functioning of a system requires invalidation of the branch predictor when there are changes to the instructions in memory, the invalidate entire instruction cache operation also causes an invalidate entire branch predictor array operation.

Branch predictor maintenance operations and the memory order model

The following rule describes the effect of the memory order model on the branch predictor maintenance operations:

- Any invalidation of the branch predictor is guaranteed to take effect only after one of the following:
  - — execution of a `ISB` instruction
  - — taking an exception
  - — return from an exception.

Therefore, if a branch instruction appears between an invalidate branch prediction instruction and an ISB operation, exception entry or exception return, it is UNPREDICTABLE whether the branch instruction is affected by the invalidate. Software must avoid this ordering of instructions, because it might lead to UNPREDICTABLE behavior.

The branch predictor maintenance operations must be used to invalidate entries in the branch predictor after any of the following events:

- enabling or disabling the MMU
- writing new data to instruction locations
- writing new mappings to the translation tables
- changes to the TTBR0, TTBR1, or TTBCR registers, unless accompanied by a change to the ContextID or the FCSE ProcessID.

Failure to invalidate entries might give UNPREDICTABLE results, caused by the execution of old branches.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

分岐予測器アレイからの MVA の無効化操作は、分岐命令のアドレスに対して実行されます。これには、現在のシステム ASID と、必要となる VA から PA への変換の一部としてどのラインが影響を受けるかが決定するときのセキュリティ状態が含まれます。セキュリティ拡張機能が実装されている場合のみ、セキュリティ状態のチェックが実行されます。MVA による無効化操作は、他の分岐予測器エントリに影響を与える可能性があります。

注

アーキテクチャは、無効化操作が適用される分岐予測器内のアドレスの範囲を可視にしません。これは、MVA 無効化命令で使用されるアドレスを、無効化される分岐のアドレスにする必要があることを意味します。

システムを正常に機能させるために、メモリ内の命令が変更された時点で分岐予測器を無効化する必要がある場合は、命令キャッシュ全体の無効化操作によっても分岐予測器全体の無効化操作が実行されます。

分岐予測器の保守操作とメモリオーダ モデル

次の規則は、分岐予測器保守操作に対するメモリオーダ モデルの影響を示しています。

- 分岐予測器の無効化は、次のいずれかの後でのみ、有効であることが保証されます。
  - – `ISB` 命令の実行
  - – 例外の取得
  - – 例外からの復帰

したがって、分岐命令が分岐予測の無効化命令と、ISB 操作、例外の開始、または例外からの復帰との間に出現する場合は、その分岐命令が無効化の影響を受けるかどうかは予測不能です。命令がこのような順序であると予測不能な動作を導く可能性があるため、ソフトウェアで回避する必要があります。

次のいずれかのイベントの後では、分岐予測器の保守操作を使用して、分岐予測器のエントリを無効化する必要があります。

- MMU の稼動または非稼動
- 命令位置への新しいデータの書き込み
- 変換テーブルへの新しいマッピングの書き込み
- ContextID または FCSE ProcessID の変更を伴わない、TTBR0、TTBR1、または TTBCR レジスタの変更

エントリの無効化に失敗した場合は、古い分岐の実行により、結果が予測不能になる可能性があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In ARMv7, there is no requirement to use the branch predictor maintenance operations to invalidate the branch predictor after:

- changing the ContextID or FCSE ProcessID.
- a cache operation that is identified as also flushing the branch target cache, see either:
  - *CP15 c7, Cache and branch predictor maintenance functions* on page B3-126 for a VMSA implementation
  - *CP15 c7, Cache and branch predictor maintenance functions* on page B4-68 for a PMSA implementation.

Note

In ARMv6, the branch predictor must be invalidated after a change to the ContextID or FCSE ProcessID, see *c13, Context ID support* on page AppxG-54.

Behavior of the branch predictors at reset

In ARMv7:

- If branch predictors are not architecturally invisible the branch prediction logic is disabled at reset.
- An implementation can require the use of a specific branch predictor initialization routine to invalidate its storage array before it is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, but the routine must be documented clearly as part of the documentation of the device.
- ARM recommends that whenever an invalidation routine is required, it is based on the ARMv7 branch predictor maintenance operations.

When it is enabled the state of the branch predictor logic is UNPREDICTABLE if the appropriate initialization routine has not been performed.

Similar rules apply:

- to cache behavior, see *Behavior of the caches at reset* on page B2-6
- on an ARMv7-A implementation, to TLB behavior, see *TLB behavior at reset* on page B3-55.

B2.2.7 Ordering of cache and branch predictor maintenance operations

The following rules describe the effect of the memory order model on the cache and branch predictor maintenance operations:

- All cache and branch predictor maintenance operations are executed, relative to each other, in program order.
- On an ARMv7-A implementation, where a cache or branch predictor maintenance operation appears in program order before a change to the translation tables, the cache or branch predictor maintenance operation is guaranteed to take place before the change to the translation tables is visible.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARMv7 では、次の操作の後で、分岐予測器の保守操作を使用して分岐予測器を無効化する要件はありません。

- ContextID または FCSE ProcessID の変更
- 分岐ターゲットキャッシュのフラッシュとしても識別されるキャッシュ操作。次のセクションを参照して下さい。
  - *CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126（VMSA 実装の場合）
  - *CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68 （PMSA 実装の場合）

注

ARMv6 では、ContextID または FCSE ProcessID の変更後に分岐予測器を無効化する必要があります。*c13、コンテキスト ID のサポート* P. 付録 G-54 を参照して下さい。

リセット時の分岐予測器の動作

ARMv7 では、次の規則が適用されます。

- 分岐予測器がアーキテクチャ上不可視になっている場合は、リセット時に分岐予測ロジックが非稼動にされます。
- 実装では、特定の分岐予測器初期化ルーチンを使用して、分岐予測器を稼動する前にその記憶域アレイを無効にすることを必須とできます。要求される初期化ルーチンの正確な形式は実装定義ですが、このルーチンはデバイスの説明書の一部として明確に文書化する必要があります。
- 無効化ルーチンが必要な場合は、ARMv7 の分岐予測器保守操作に基づくことをお勧めします。

適切な初期化ルーチンが実行されなかった場合は、分岐予測器が稼動されたときの分岐予測ロジックの状態が予測不能になります。

次の動作にも、同様の規則が適用されます。

- キャッシュの動作。*リセット時のキャッシュの動作* P. B2-6 を参照して下さい。
- ARMv7-A 実装での TLB の動作。*リセット時の TLB の動作* P. B3-55 を参照して下さい。

B2.2.7 キャッシュの保守操作と分岐予測器の保守操作との順序付け

次の規則は、キャッシュの保守操作と分岐予測器の保守操作とに対するメモリオーダ モデルの影響を示しています。

- すべてのキャッシュの保守操作と分岐予測器の保守操作は、相対的にプログラムでの順序で実行されます。
- ARMv7-A 実装で、キャッシュの保守操作または分岐予測器の保守操作がプログラムの順序で変換テーブルの変更よりも前に出現する場合、変換テーブルの変更が可視になる前に、キャッシュの保守操作または分岐予測器の保守操作が実行されることが保証されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- On an ARMv7-A implementation, where a change of the translation tables appears in program order before a cache or branch predictor maintenance operation, that change is guaranteed to be visible only after the sequence outlined in *TLB maintenance operations and the memory order model* on page B3-59 is executed.
- A DMB instruction causes the effect of all data cache or unified cache maintenance operations appearing in program order before the DMB to be visible to all explicit load and store operations appearing in program order after the DMB.

  It also ensures that the effects of any data cache or unified cache maintenance operations appearing in program order before the DMB are observable by any observer in the same required shareability domain before any data cache or unified cache maintenance or explicit memory operations appearing in program order after the DMB are observed by the same observer. Completion of the DMB does not guarantee the visibility of all data to other observers. For example, all data might not be visible to a translation table walk, or to instruction fetches.
- A DSB causes the completion of all cache maintenance operations appearing in program order before the DSB instruction.
- An ISB instruction or an exception entry or a return from exception causes the effect of all branch predictor maintenance operations appearing in program order before the ISB instruction, exception entry or exception return to be visible to all instructions after the ISB instruction, exception entry or exception return.
- Any data cache or unified cache maintenance operation by MVA must be executed in program order relative to any explicit load or store on the same processor to an address covered by the MVA of the cache operation. The order of memory accesses that result from the cache maintenance operation, relative to any other memory accesses, are subject to the memory ordering rules. For more information, see *Ordering requirements for memory accesses* on page A3-45.
- There is no restriction on the ordering of data cache or unified cache maintenance operations by MVA relative to any explicit load or store on the same processor where the address of the explicit load or store is not covered by the MVA of the cache operation. Where the ordering must be restricted, a DMB instruction must be inserted to enforce ordering.
- There is no restriction on the ordering of a data cache or unified cache maintenance operation by set/way relative to any explicit load or store on the same processor. Where the ordering must be restricted, a DMB instruction must be inserted to enforce ordering.
- The execution of a data cache or unified cache maintenance operation by set/way might not be visible to other observers in the system until after a DSB instruction is executed.
- The execution of an instruction cache maintenance operation is guaranteed to be complete only after the execution of a DSB instruction.
- The completion of an instruction cache maintenance operation is guaranteed to be visible to the instruction fetch only after the execution of an ISB instruction or an exception entry or return from exception.

The last two points mean that the sequence of cache cleaning operations for a line of self-modifying code on a uniprocessor system is:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- ARMv7-A 実装で、変換テーブルの変更がプログラムの順序でキャッシュの保守操作または分岐予測器の保守操作よりも前に出現する場合、*TLB 保守操作とメモリオーダ モデル* P. B3-59 で説明されているシーケンスの実行後でなければ、その変更が可視にならないことが保証されます。
- DMB 命令がある場合、プログラムの順序で DMB の前に出現するすべてのデータまたは統一キャッシュの保守操作の影響は、プログラムの順序で DMB の後に出現する、すべての明示的なロード操作およびストア操作から可視です。

  また、プログラムの順序で DMB の前に出現する任意のデータまたは統一キャッシュ保守操作の影響は、プログラムの順序で DMB の後に出現する任意のデータまたは統一キャッシュ保守操作、あるいは明示的なメモリ操作が、必要な共有可ドメイン内のオブザーバによって観測される前に、その同じオブザーバによって観測可能になることも保証されます。DMB が完了しても、すべてのデータが他のオブザーバから可視であることは保証されません。例えば、すべてのデータが変換テーブルウォークまたは命令フェッチから可視でない可能性があります。
- DSB 命令によって、プログラムの順序で DSB の前に出現するすべてのキャッシュ保守操作が完了します。
- ISB 命令、例外の開始、または例外からの復帰がある場合、プログラムの順序で ISB 命令、例外の開始、または例外からの復帰の前に出現するすべての分岐予測器の保守操作の影響は、ISB 命令、例外の開始、または例外からの復帰の後に出現するすべての命令から可視です。
- MVA によるデータまたは統一キャッシュの保守操作は、そのキャッシュ操作の MVA によってカバーされるアドレスに対して同じプロセッサにより実行される、明示的なロードまたはストアと、相対的にプログラムの順序で実行する必要があります。キャッシュの保守操作によって生じるメモリアクセスと、その他のすべてのメモリアクセスとの相対的な順序は、メモリ順序の規則の対象となります。詳細については、*メモリアクセスの順序付けの要件* P. A3-45 を参照して下さい。
- MVA によるデータキャッシュまたは統一キャッシュの保守操作と、同じプロセッサにより行われる明示的なロードまたはストアについて、そのロードまたはストアがキャッシュ操作の MVA によりカバーされない場合は、相対的な順序に関する制約はありません。順序付けを制限する必要がある場合は、DMB 命令を挿入して順序付けを強制する必要があります。
- セット / ウェイによるデータキャッシュまたは統一キャッシュの保守操作と、同じプロセッサにより行われる明示的なロードまたはストアとの間には、相対的な順序に関する制約はありません。順序付けを制限する必要がある場合は、DMB 命令を挿入して順序付けを強制する必要があります。
- セット / ウェイによるデータまたは統一キャッシュの保守操作の実行は、DSB 命令が実行されるまで、システムの他のオブザーバに対して可視でない可能性があります。
- 命令キャッシュの保守操作の実行は、DSB 命令の実行後にのみ、完了したことが保証されます。
- 命令キャッシュの保守操作の完了は、ISB 命令の実行、例外の開始、または例外からの復帰の後にのみ、命令フェッチから可視であることが保証されます。

最後の 2 つのポイントは、ユニプロセッサシステム上の自己変更コード行に関するキャッシュクリーニング操作のシーケンスが次のようになることを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
; Enter this code with <Rx> containing the new 32-bit instruction. Use STRH in the first
; line instead of STR for a 16-bit instruction.
        STR     <Rx>, [instruction location]
        Clean data cache by MVA to point of unification [instruction location]
        DSB                     ; Ensures visibility of the data cleaned from the data cache
        Invalidate instruction cache by MVA [instruction location]
        Invalidate BTC entry by MVA [instruction location]
        DSB                     ; Ensures completion of the instruction cache invalidation
        ISB
```

B2.2.8 Multiprocessor effects on cache maintenance operations

This section describes the multiprocessor effects on cache maintenance operations for the base ARMv7 architecture and the base ARMv7 architecture with Multiprocessing Extensions.

Base ARMv7 architecture

The base ARMv7 architecture defines that all cache maintenance operations apply only to the caches directly attached to the processor on which the operation is executed. There is no requirement that cache maintenance operations influence all processors with which the data can be shared.

In porting an architecturally portable multiprocessor operating system to ARMv7, when a cache maintenance operation is performed, *Inter-Processor Interrupts* (IPIs) must be used to inform other processors in a multiprocessor configuration that they must perform the equivalent operation.

Multiprocessing Extensions

To improve the implementation of multiprocessor systems, a set of extensions to ARMv7, called the Multiprocessing Extensions, has been introduced. These expand the role of cache and branch predictor maintenance operations in the multiprocessing system. For the VMSA architecture, the Multiprocessing Extensions also extend the role of TLB operations. For more information see *Multiprocessor effects on TLB maintenance operations* on page B3-62.

The extensions can be implemented in a uniprocessor system with no hardware support for cache coherency. In such a system, the Inner Shareable and Outer Shareable domains would be limited to being the single processor, and all instructions defined to apply to the Inner Shareable domains behave as aliases of the local operations.

Data and Unified cache operations to the point of coherency

The following instructions have an effect on data and unified caches to the point of coherency, and must affect the caches of other processors in the shareability domain described by the shareability attributes of the MVA passed with the instruction:

- invalidate data, or unified, cache line by MVA to the point of coherency (DCIMVAC)
- clean data, or unified, cache line by MVA to the point of coherency (DCCMVAC)
- clean and invalidate data (or unified) cache line by MVA to the point of coherency (DCCIMVAC).

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
; Enter this code with <Rx> containing the new 32-bit instruction. Use STRH in the first
; line instead of STR for a 16-bit instruction.
        STR     <Rx>, [instruction location]
        Clean data cache by MVA to point of unification [instruction location]
        DSB                     ; Ensures visibility of the data cleaned from the data cache
        Invalidate instruction cache by MVA [instruction location]
        Invalidate BTC entry by MVA [instruction location]
        DSB                     ; Ensures completion of the instruction cache invalidation
        ISB
```


B2.2.8 キャッシュの保守操作に対するマルチプロセッサの影響

このセクションでは、基本 ARMv7 アーキテクチャおよびマルチプロセッシング拡張機能が実装された基本 ARMv7 アーキテクチャのキャッシュの保守操作に対する、マルチプロセッサの影響について説明します。

基本 ARMv7 アーキテクチャ

基本 ARMv7 アーキテクチャでは、すべてのキャッシュの保守操作が、その操作が実行されるプロセッサに直接接続されているキャッシュに対してのみ適用されることが定義されています。キャッシュの保守操作がデータを共有可能なすべてのプロセッサに影響を及ぼす要件はありません。

アーキテクチャ上移植可能なマルチプロセッサ オペレーティングシステムの ARMv7 への移植では、キャッシュ保守操作が実行されるとき、*内部プロセッサ割り込み* (IPI) を使用して、マルチプロセッサ構成に含まれる他のプロセッサに対して、同等の操作を実行する必要があることを通知する必要があります。

マルチプロセッシング拡張機能

マルチプロセッサシステムの実装を強化するために、マルチプロセッシング拡張機能と呼ばれる ARMv7 に対する一連の拡張機能が導入されました。これらの拡張機能によって、マルチプロセッシングシステムでのキャッシュと分岐予測器の保守操作の役割が拡張されます。VMSA アーキテクチャの場合は、マルチプロセッシング拡張機能によって TLB 操作の役割も拡張されます。詳細については、*TLB 保守操作に対するマルチプロセッサの影響* P. B3- 62 を参照して下さい。

これらの拡張機能は、キャッシュコヒーレンシのハードウェアサポートのないユニプロセッサシステムに実装することができます。このようなシステムでは、内部共有可能ドメインと外部共有可能ドメインがシングルプロセッサに限定され、内部共有可能ドメインに適用するために定義されたすべての命令がローカル操作のエイリアスとして動作します。

コヒーレンシ時点までのデータおよび統一キャッシュの操作

次に示す命令は、コヒーレンシ時点までのデータキャッシュと統一キャッシュに影響を与えるだけでなく、命令と一緒に渡された MVA の共有可能属性で示される共有可能ドメインに含まれる他のプロセッサのキャッシュに影響を及ぼす必要があります。

- コヒーレンシ時点までの、MVA によるデータまたは統一キャッシュラインの無効化 (DCIMVAC)
- コヒーレンシ時点までの、MVA によるデータまたは統一キャッシュラインのクリーニング (DCCMVAC)
- コヒーレンシ時点までの、MVA によるデータまたは統一キャッシュラインのクリーニングと無効化 (DCCIMVAC)

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B2-1 shows, for these instructions, the minimum set of processors that they affect, and the earliest point that the operations occur to depends upon the shareability attribute of the address being used.

Table B2-1 Processors affected by Data and Unified cache operations

Shareability	Processors affected	Point that the operations occur to
Non-shareable	The processor executing the instruction	Point of coherency of the entire system
Inner Shareable	All processors in the same Inner Shareable shareability domain as the processor executing the instruction	Point of coherency of the entire system
Outer Shareable	All processors in the same Outer shareable shareability domain as the processor executing the instruction	Point of coherency of the entire system

Address based cache maintenance operations not to the point of coherency

The following operations are redefined in the Multiprocessing Extensions:

- Clean data, or unified, cache line by MVA to the point of unification (DCCMVAU)
- Invalidate instruction cache line by MVA to point of unification (ICIMVAU)
- Invalidate MVA from branch predictor array (BPIMVA)

Table B2-2 shows, for these instructions, the minimum set of processors that they effect, and the earliest point that the operations occur to depends upon the shareability attribute of the address being used.

Table B2-2 Processors affected byAddress based cache maintenance operations

Shareability of the Address	Processors affected	Point that the operations occur to
Non-Shareable	The processor executing the instruction	To the point of unification of instruction cache fills, data cache fills and writebacks, and translation table walks on the processor executing the instruction
Inner Shareable or Outer shareable	All processors in the same Inner Shareable shareability domain as the processor executing the instruction	To the point of unification of instruction cache fills, data cache fills and writebacks, and translation table walks of all processors in the same Inner Shareable shareability domain as the processor executing the instruction

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 B2-1 は、これらの命令に関して、影響を受けるプロセッサの最小セットと、使用されるアドレスの共有可能属性に基づいて操作が実行される最も早い時点を示しています。

表 B2-1 データキャッシュと統一キャッシュの操作の影響を受けるプロセッサ

共有可能性	影響を受けるプロセッサ	操作が実行される時点
共有不可	命令を実行中のプロセッサ	システム全体のコヒーレンシ時点
内部共有可能	命令を実行中のプロセッサと同じ内部共有可能な共有可能ドメインの、すべてのプロセッサ	システム全体のコヒーレンシ時点
外部共有可能	命令を実行中のプロセッサと同じ外部共有可能な共有可能ドメイン内の、すべてのプロセッサ	システム全体のコヒーレンシ時点

コヒーレンシ時点まで行わないアドレスベースのキャッシュ保守操作

次の操作は、マルチプロセッシング拡張機能で再定義されます。

- コヒーレンシ時点までの、MVA によるデータまたは統一キャッシュラインのクリーニング (DCCMVAU)
- 統一時点までの、MVA による命令キャッシュラインの無効化 (ICIMVAU)
- 分岐予測器アレイからの MVA の無効化 (BPIMVA)

これらの命令に関して、影響を受けるプロセッサの最小セットと、使用されるアドレスの共有可能属性に基づいて操作が実行される最も早い時点とを、表 B2-2 に示します。

表 B2-2 アドレスベースのキャッシュ保守操作の影響を受けるプロセッサ

アドレスの共有可能性	影響を受けるプロセッサ	操作がどの時点まで行われるか
共有不可	命令を実行中のプロセッサ	命令を実行中のプロセッサ上の命令キャッシュフィル、データキャッシュフィルとライトバック、および変換テーブルウォークの統一時点まで
内部共有可能または外部共有可能	命令を実行中のプロセッサと同じ内部共有可能な共有可能ドメインの、すべてのプロセッサ	命令を実行中のプロセッサと同じ内部共有可能な共有可ドメインの、すべてのプロセッサの命令キャッシュフィル、データキャッシュフィルとライトバック、および変換テーブルウォークの統一時点まで

注

確実に影響を受けるプロセッサの数が、実行中のプロセッサを含む内部共有可能な共有可能ドメインより多くなることはありません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

The set of processors that is guaranteed to be affected is never greater than the Inner Shareable shareability domain containing the executing processor.

Entire and set/way based cache maintenance operations

This section describes the Local and Inner Shareable instructions for entire and set/way based cache maintenance operations:

Local instructions

The following instructions are only guaranteed to apply to the caches of the processor that the instructions are run on:

- Invalidate entire instruction cache (ICIALLU)
- Invalidate entire branch predictor array (BPIALL)
- Clean and Invalidate data or unified cache line by set/way (DCCISW)
- Clean data or unified cache line by set/way (DCCSW)
- Invalidate data or unified cache line by set/way (DCISW).

These operations have an effect on the processor executing the instruction.

These operations are functionally unchanged from the base architecture.

Inner Shareable instructions

The following instructions can be applied to the caches of all processors in the same Inner Shareable shareability domain as the processor executing the instruction:

- Invalidate entire branch predictor array Inner Shareable (BPIALLIS)
- Invalidate entire instruction cache Inner Shareable (ICIALLUIS).

ICIALLUIS automatically performs the BPIALLIS function, in the same way as ICIALLU automatically performs the BPIALL function.

These operations have an effect to the point of unification of instruction cache fills, data cache fills and writebacks, and translation table walks of all processors in the same Inner Shareable shareability domain.

These instructions complement the ICIALLU and BPIALL instructions defined in the base ARMv7 architecture, and extend them to the same Inner Shareable shareability domain.

Inner Shareable instructions encodings:

- ICIALLUIS is encoded as `MCR p15, 0, <Rt>, c7, c1, 0`
- BPIALLIS is encoded as `MCR p15, 0, <Rt>, c7, c1, 6`

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

キャッシュ全体の保守操作とセット / ウェイベースのキャッシュ保守操作

このセクションでは、キャッシュ全体の保守操作とセット / ウェイベースのキャッシュ保守操作に関する、ローカル命令と内部共有可能命令について説明します。

ローカル命令

次の命令は、命令を実行するプロセッサのキャッシュにのみ適用されることが保証されています。

- 命令キャッシュ全体の無効化 (ICIALLU)
- 分岐予測器アレイ全体の無効化 (BPIALL)
- セット / ウェイによるデータまたは統一キャッシュラインのクリーニングと無効化 (DCCISW)
- セット / ウェイによるデータまたは統一キャッシュラインのクリーニング (DCCSW)
- セット / ウェイによるデータまたは統一キャッシュラインの無効化 (DCISW)

これらの操作は、命令を実行中のプロセッサに影響を与えます。

これらの操作の機能は、基本アーキテクチャから変更されていません。

内部共有可能命令

次の命令は、命令を実行中のプロセッサと同じ内部共有可能な共有可能ドメインの、すべてのプロセッサのキャッシュに適用できます。

- 内部共有可能な分岐予測器アレイ全体の無効化 (BPIALLIS)
- 内部共有可能な命令キャッシュ全体の無効化 (ICIALLUIS)

ICIALLUIS は、ICIALLU が自動的に BPIALL 関数を実行するのと同様に、自動的に BPIALLIS 関数を実行します。

これらの操作は、同じ内部共有可能な共有可能ドメインに存在する、すべてのプロセッサの命令キャッシュフィル、データキャッシュフィルとライトバック、および変換テーブルウォークの統一時点まで効力を持ちます。

これらの命令は、基本 ARMv7 アーキテクチャで定義された ICIALLU および BPIALL 命令を補完し、同じ内部共有可能な共有可能ドメインへ拡張します。

内部共有可能命令のエンコードは次のとおりです。

- ICIALLUIS は、`MCR p15, 0, <Rt>, c7, c1, 0` としてエンコードされます。
- BPIALLIS は、`MCR p15, 0, <Rt>, c7, c1, 6` としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.2.9 System-level caches

The system-level architecture might define further aspects of the software view of caches and the memory model that are not defined by the ARMv7 processor architecture. These aspects of the system-level architecture can affect the requirements for software management of caches and coherency. For example, a system design might introduce additional levels of caching that cannot be managed using the CP15 maintenance operations defined by the ARMv7 architecture. Typically, such caches are referred to as *system caches* and are managed through the use of memory-mapped operations. The ARMv7 architecture does not forbid the presence of system caches that are outside the scope of the architecture, but ARM strongly recommends the following for any such cache:

- Physical, rather than virtual, addresses are used for address-based cache maintenance operations.
- Any IMPLEMENTATION DEFINED system cache maintenance operations include as a minimum the set of functions defined by *ARMv7 cache maintenance operations* on page B2-13, with the number of levels of system cache operated on by these cache maintenance operations being IMPLEMENTATION DEFINED.
- Where possible, such system caches are included in the caches affected by the architecturally-defined CP15 cache maintenance operations, so that the architecturally-defined software sequences for managing the memory model and coherency are sufficient for managing all caches in the system.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B2.2.9 システムレベルのキャッシュ

システムレベルのアーキテクチャでは、ソフトウェアから見えるキャッシュの新しい側面と、ARMv7 プロセッサアーキテクチャで定義されていないメモリモデルを定義することができます。このようなシステムレベルのアーキテクチャの側面が、キャッシュとコヒーレンシのソフトウェア管理に関する要件に影響を与える可能性があります。例えば、システム設計で、ARMv7 アーキテクチャで定義されている CP15 保守操作では管理できない新しいキャッシュレベルを導入する場合です。このようなキャッシュは一般にシステムキャッシュと呼ばれ、メモリマップされた操作を使用して管理されます。ARMv7 アーキテクチャでは、アーキテクチャの範囲外にあるシステムキャッシュの存在が禁止されていませんが、このようなキャッシュには次のような方針を強くお勧めします。

- アドレスベースのキャッシュ保守操作では、仮想アドレスではなく物理アドレスを使用します。
- 実装定義のシステムキャッシュ保守操作には、*ARMv7 のキャッシュ保守操作* P. B2-13 で定義されている機能の最小セットと、実装定義のキャッシュ保守操作によって処理されるシステムキャッシュのレベル数を含めます。
- 可能であれば、このようなシステムキャッシュを、アーキテクチャ定義の CP15 キャッシュ保守操作の影響を受けるキャッシュ内に含めることによって、メモリモデルとコヒーレンシを管理するためのアーキテクチャ定義のソフトウェアシーケンスにより、システムのすべてのキャッシュを管理することができます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.3 IMPLEMENTATION DEFINED memory system features

ARMv7 reserves space in the SCTLR for use with IMPLEMENTATION DEFINED features of the cache, and other IMPLEMENTATION DEFINED features of the memory system architecture.

In particular, in ARMv7 the following memory system features are IMPLEMENTATION DEFINED:

- Cache lockdown, see *Cache lockdown* on page B2-8.
- In VMSAv7, TLB lockdown, see *TLB lockdown* on page B3-56.
- *Tightly Coupled Memory* (TCM) support, including any associated DMA scheme. The TCM Type Register, TCMTR is required in all implementations, and if no TCMs are implemented this must be indicated by the value of this register.

  Note

  For details of the optional TCMs and associated DMA scheme in ARMv6 see *Tightly Coupled Memory (TCM) support* on page AppxG-23.

B2.3.1 ARMv7 CP15 register support for IMPLEMENTATION DEFINED features

The ARMv7 CP15 registers implementation includes the following support for IMPLEMENTATION DEFINED features of the memory system:

- The TCM Type Register, TCMTR, in CP15 c0, must be implemented. The following conditions apply to this register:
  - If no TCMs are implemented, the TCMTR indicates zero-size TCMs. For more information see *c0, TCM Type Register (TCMTR)* on page B3-85 (for a VMSA implementation) or *c0, TCM Type Register (TCMTR)* on page B4-35 (for a PMSA implementation).
  - If bits [31:29] are 0b100, the format of the rest of the register format is IMPLEMENTATION DEFINED. This value indicates that the implementation includes TCMs that do not follow the ARMv6 usage model. Other fields in the register might give more information about the TCMs.

  For more information, see:
  - *c0, TCM Type Register (TCMTR)* on page B3-85, for a VMSA implementation
  - *c0, TCM Type Register (TCMTR)* on page B4-35, for a PMSA implementation.
- The CP15 c9 encoding space with `<CRm>` = {0-2,5-7} is IMPLEMENTATION DEFINED for all values of `<opc2>` and `<opc1>`. This space is reserved for branch predictor, cache and TCM functionality, for example maintenance, override behaviors and lockdown. It permits:
  - ARMv6 backwards compatible schemes
  - alternative schemes.

  For more information, see:
  - *CP15 c9, Cache and TCM lockdown registers and performance monitors* on page B3-141, for a VMSA implementation
  - *CP15 c9, Cache and TCM lockdown registers and performance monitors* on page B4-74, for a PMSA implementation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.3 実装定義のメモリシステムの機能

ARMv7 では、キャッシュの実装定義の機能と、メモリシステムのアーキテクチャに含まれる他の実装定義の機能で使用するために、SCTLR 内の空間が予約されています。

具体的には、ARMv7 では、次のメモリシステム機能が実装定義です。

- キャッシュのロックダウン。*キャッシュロックダウン* P. B2-8 を参照して下さい。
- VMSAv7 での TLB ロックダウン。*TLB ロックダウン* P. B3-56 を参照して下さい。
- 関連するすべての DMA 方式を含む、*密結合メモリ* (TCM) のサポート。TCM タイプレジスタの TCMTR がすべての実装に必要です。TCM が実装されていない場合は、このレジスタの値により、そのことが示されている必要があります。

  注

  ARMv6 におけるオプションの TCM、および関連する DMA 方式の詳細については、*密結合メモリ (TCM) のサポート* P. 付録 G-23 を参照して下さい。

B2.3.1 ARMv7 CP15 レジスタによる実装定義の機能のサポート

ARMv7 CP15 レジスタの実装には、メモリシステムの実装定義の機能に対する次のようなサポートが含まれます。

- CP15 c0 内の TCM タイプレジスタ（TCMTR）を実装する必要があります。このレジスタには、次の条件が適用されます。
  - TCM が実装されていない場合は、TCMTR がサイズ 0 の TCM を示します。詳細については、*c0、TCM タイプレジスタ (TCMTR)* P. B3-85（VMSA 実装の場合）または *c0、TCM タイプレジスタ (TCMTR)* P. B4-35（PMSA 実装の場合）を参照して下さい。
  - ビット [31:29] が 0b100 の場合、残りのレジスタフォーマットは実装定義です。この値は、ARMv6 使用モデルに従っていない TCM が実装に含まれていることを意味します。レジスタの他のフィールドで、TCM の詳細が提供される場合があります。

  詳細については、次の各セクションを参照して下さい。
  - *c0、TCM タイプレジスタ (TCMTR)* P. B3-85（VMSA 実装の場合）
  - *c0、TCM タイプレジスタ (TCMTR)* P. B4-35（PMSA 実装の場合）
- `<CRm>` = {0-2,5-7} の CP15 c9 エンコード空間は、`<opc2>` と `<opc1>` のすべての値に対して実装定義です。この空間は、分岐予測器、キャッシュ、および TCM の機能（保守、動作のオーバーライド、ロックダウンなど）のために予約されています。この空間では、次の方式が許可されます。
  - ARMv6 と下位互換の方式
  - 代替方式

  詳細については、次の各セクションを参照して下さい。
  - *CP15 c9、キャッシュおよび TCM ロックダウンレジスタとパフォーマンスモニタ* P. B3-141（VMSA 実装の場合）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- In a VMSAv7 implementation, part of the CP15 c10 encoding space is IMPLEMENTATION DEFINED and reserved for TLB functionality, see *TLB lockdown* on page B3-56.
- The CP15 c11 encoding space with `<CRm>` = {0-8,15} is IMPLEMENTATION DEFINED for all values of `<opc2>` and `<opc1>`. This space is reserved for DMA operations to and from the TCMs It permits:
  - — an ARMv6 backwards compatible scheme
  - — an alternative scheme.

  For more information, see:
  - — *CP15 c11, Reserved for TCM DMA registers* on page B3-147, for a VMSA implementation
  - — *CP15 c11, Reserved for TCM DMA registers* on page B4-75, for a PMSA implementation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- — *CP15 c9、キャッシュおよび TCM ロックダウンレジスタとパフォーマンスモニタ* P. B4-74（PMSA 実装の場合）

- VMSAv7 実装では、CP15 c10 エンコード空間の一部が実装定義であり、TLB 機能のために予約されています。*TLB ロックダウン* P. B3-56 を参照して下さい。
- `<CRm>` = {0-8,15} の CP15 c11 エンコード空間は、`<opc2>` と `<opc1>` のすべての値に対して実装定義です。この空間は、TCM との間での DMA 操作用に予約されています。この空間では次の方式が許可されます。
  - — 1 つの ARMv6 と下位互換の方式
  - — 1 つの代替方式

  詳細については、次の各セクションを参照して下さい。
  - — *CP15 c11、TCM DMA 用に予約されているレジスタ* P. B3-147（VMSA 実装の場合）
  - — *CP15 c11、TCM DMA 用に予約されているレジスタ* P. B4-75（PMSA 実装の場合）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.4 Pseudocode details of general memory system operations

This section contains pseudocode describing general memory operations, in the subsections:

- *Memory data type definitions*.
- *Basic memory accesses* on page B2-30.
- *Interfaces to memory system specific pseudocode* on page B2-30.
- *Aligned memory accesses* on page B2-31
- *Unaligned memory accesses* on page B2-32
- *Reverse endianness* on page B2-34
- *Exclusive monitors operations* on page B2-35
- *Access permission checking* on page B2-37
- *Default memory access decode* on page B2-37
- *Data Abort exception* on page B2-39.

The pseudocode in this section applies to both VMSA and PMSA implementations. Additional pseudocode for memory operations is given in:

- *Pseudocode details of VMSA memory system operations* on page B3-156
- *Pseudocode details of PMSA memory system operations* on page B4-79.

B2.4.1 Memory data type definitions

The following data type definitions are used by the memory system pseudocode functions:

```
// Types of memory

enumeration MemType {MemType_Normal, MemType_Device, MemType_StronglyOrdered};

// Memory attributes descriptor

type MemoryAttributes is (
    MemType type,
    bits(2) innerattrs,  // '00' = Non-cacheable; '01' = WBWA; '10' = WT; '11' = WBnWA
    bits(2) outerattrs,  // '00' = Non-cacheable; '01' = WBWA; '10' = WT; '11' = WBnWA
    boolean shareable,
    boolean outershareable
)

// Physical address type, with extra bits used by some VMSA features

type FullAddress is (
    bits(32) physicaladdress,
    bits(8)  physicaladdressext,
    bit      NS                  // '0' = Secure, '1' = Non-secure
)

// Descriptor used to access the underlying memory array

type AddressDescriptor is (
    MemoryAttributes memattrs,
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.4 一般的なメモリシステム操作の擬似コードの詳細

このセクションでは、一般的なメモリ操作に関する擬似コードについて、次に示すサブセクションで説明します。

- *メモリのデータ型の定義*
- *基本的なメモリアクセス* P. B2-30
- *メモリシステム固有の擬似コードに対するインタフェース* P. B2-30
- *アライン メモリアクセス* P. B2-31
- *アンアラインドメモリアクセス* P. B2-32
- *エンディアン形式の反転* P. B2-34
- *排他モニタ操作* P. B2-35
- *アクセス許可のチェック* P. B2-37
- *デフォルトのメモリアクセス デコード* P. B2-37
- *データアボート例外* P. B2-39

このセクションで説明する擬似コードは、VMSA 実装と PMSA 実装の両方に適用されます。メモリ操作に関する他の擬似コードについては、次のセクションを参照して下さい。

- *VMSA メモリシステム操作の擬似コードの詳細* P. B3-156
- *PMSA メモリシステム操作の擬似コードの詳細* P. B4-79

B2.4.1 メモリのデータ型の定義

次のデータ型の定義が、メモリシステム擬似コード関数によって使用されます。

```
// Types of memory

enumeration MemType {MemType_Normal, MemType_Device, MemType_StronglyOrdered};

// Memory attributes descriptor

type MemoryAttributes is (
    MemType type,
    bits(2) innerattrs,  // '00' = Non-cacheable; '01' = WBWA; '10' = WT; '11' = WBnWA
    bits(2) outerattrs,  // '00' = Non-cacheable; '01' = WBWA; '10' = WT; '11' = WBnWA
    boolean shareable,
    boolean outershareable
)

// Physical address type, with extra bits used by some VMSA features

type FullAddress is (
    bits(32) physicaladdress,
    bits(8)  physicaladdressext,
    bit      NS                  // '0' = Secure, '1' = Non-secure
)

// Descriptor used to access the underlying memory array

type AddressDescriptor is (
    MemoryAttributes memattrs,
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    FullAddress      paddress
)

// Access permissions descriptor

type Permissions is (
    bits(3) ap,   // Access Permission bits
    bit     xn    // Execute Never bit
)
```

B2.4.2 Basic memory accesses

The _Mem[] function performs single-copy atomic, aligned, little-endian memory accesses to the underlying physical memory array of bytes:

```
bits(8*size) _Mem[AddressDescriptor memaddrdesc, integer size]
    assert size == 1 || size == 2 || size == 4 || size == 8;

_Mem[AddressDescriptor memaddrdesc, integer size] = bits(8*size) value
    assert size == 1 || size == 2 || size == 4 || size == 8;
```

This function addresses the array using memaddrdesc.paddress,that supplies:

- A 32-bit physical address.
- An 8-bit physical address extension, that is treated as additional high-order bits of the physical address. This extension is always 0b00000000 in the PMSA.
- A single NS bit to select between Secure and Non-secure parts of the array. This bit is always 0 if the Security Extensions are not implemented.

The actual implemented array of memory might be smaller than the 2^{41} bytes implied. In this case, the scheme for aliasing is IMPLEMENTATION DEFINED, or some parts of the address space might give rise to external aborts. For more information, see:

- *External aborts* on page B3-45 for a VMSA implementation
- *External aborts* on page B4-15 for a PMSA implementation.

The attributes in memaddrdesc.memattrs are used by the memory system to determine caching and ordering behaviors as described in *Memory types and attributes and the memory order model* on page A3-24.

B2.4.3 Interfaces to memory system specific pseudocode

The following functions call the VMSA-specific or PMSA-specific functions to handle Alignment faults and perform address translation.

```
// AlignmentFault()
// ================

AlignmentFault(bits(32) address, boolean iswrite)
    case MemorySystemArchitecture() of
        when MemArch_VMSA  AlignmentFaultV(address, iswrite);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    FullAddress      paddress
)

// Access permissions descriptor

type Permissions is (
    bits(3) ap,   // Access Permission bits
    bit     xn    // Execute Never bit
)
```

B2.4.2 基本的なメモリアクセス

_Mem[] 関数は、基礎となる物理メモリのバイトアレイに対して、シングルコピー アトミックでアラインされたリトルエンディアンのメモリアクセスを実行します。

```
bits(8*size) _Mem[AddressDescriptor memaddrdesc, integer size]
    assert size == 1 || size == 2 || size == 4 || size == 8;

_Mem[AddressDescriptor memaddrdesc, integer size] = bits(8*size) value
    assert size == 1 || size == 2 || size == 4 || size == 8;
```

この関数は、次のものを供給する memaddrdesc.paddress を使用してアレイのアドレスを指定します。

- 32 ビットの物理アドレス
- 物理アドレスの追加の上位ビットとして扱われる、8 ビットの物理アドレス拡張。PMSA では、この拡張は常に 0b00000000 です。
- アレイのセキュアな部分と非セキュアな部分を選択するための単一の NS ビット。セキュリティ拡張機能が実装されていない場合は、このビットは常に 0 です。

実際に実装されるメモリアレイは、暗黙の 2^{41} バイトより小さい場合があります。この場合、エリアシング方式が実装定義になるか、アドレス空間の一部で外部アボートが発生する可能性があります。詳細については、次のセクションを参照して下さい。

- *外部アボート* P. B3-45（VMSA 実装の場合）
- *外部アボート* P. B4-15 （PMSA 実装の場合）

メモリのタイプおよび属性とメモリオーダ モデル P. A3-24 で説明されているように、memaddrdesc.memattrs 内の属性は、メモリシステムでキャッシュと順序付けの動作を決定するために使用されます。

B2.4.3 メモリシステム固有の擬似コードに対するインタフェース

次の関数は、VMSA 固有または PMSA 固有の関数を呼び出して、アライメントフォールトを処理し、アドレス変換を実行します。

```
// AlignmentFault()
// ================

AlignmentFault(bits(32) address, boolean iswrite)
    case MemorySystemArchitecture() of
        when MemArch_VMSA  AlignmentFaultV(address, iswrite);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        when MemArch_PMSA  AlignmentFaultP(address, iswrite);

// TranslateAddress()
// ==================

AddressDescriptor TranslateAddress(bits(32) VA, boolean ispriv, boolean iswrite)
    case MemorySystemArchitecture() of
        when MemArch_VMSA  return TranslateAddressV(VA, ispriv, iswrite);
        when MemArch_PMSA  return TranslateAddressP(VA, ispriv, iswrite);
```

B2.4.4 Aligned memory accesses

The `MemA[]` function performs a memory access at the current privilege level, and the `MemA_unpriv[]` function performs an access that is always unprivileged. In both cases the architecture requires the access to be aligned, and in ARMv7 the function generates an Alignment fault if it is not.

Note

In versions of the architecture before ARMv7, if the SCTLR.A and SCTLR.U bits are both 0, an unaligned access is forced to be aligned by replacing the low-order address bits with zeros.

```
// MemA[]
// ======

bits(8*size) MemA[bits(32) address, integer size]
    return MemA_with_priv[address, size, CurrentModeIsPrivileged()];

MemA[bits(32) address, integer size] = bits(8*size) value
    MemA_with_priv[address, size, CurrentModeIsPrivileged()] = value;
    return;

// MemA_unpriv[]
// =============

bits(8*size) MemA_unpriv[bits(32) address, integer size]
    return MemA_with_priv[address, size, FALSE];

MemA_unpriv[bits(32) address, integer size] = bits(8*size) value
    MemA_with_priv[address, size, FALSE] = value;
    return;

// MemA_with_priv[]
// ================

// Non-assignment form

bits(8*size) MemA_with_priv[bits(32) address, integer size, boolean privileged]

    // Sort out alignment
    if address == Align(address, size) then
        VA = address;
    elsif SCTLR.A == '1' || SCTLR.U == '1' then
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
        when MemArch_PMSA  AlignmentFaultP(address, iswrite);

// TranslateAddress()
// ==================

AddressDescriptor TranslateAddress(bits(32) VA, boolean ispriv, boolean iswrite)
    case MemorySystemArchitecture() of
        when MemArch_VMSA  return TranslateAddressV(VA, ispriv, iswrite);
        when MemArch_PMSA  return TranslateAddressP(VA, ispriv, iswrite);
```


B2.4.4 アラインド メモリアクセス

`MemA[]` 関数は現在の特権レベルでメモリアクセスを実行しますが、`MemA_unpriv[]` 関数は常に非特権のアクセスを実行します。いずれの場合も、アーキテクチャ上はアラインドアクセスが要求されます。ARMv7 では、アラインドアクセスでない場合に、関数によりアライメントフォールトが発生します。

注

ARMv7 以前のバージョンのアーキテクチャでは、SCTLR.A ビットと SCTLR.U ビットの両方が 0 の場合に、アンアラインドアクセスの下位アドレスビットが強制的に 0 に置き換えられ、アラインされます。

```
// MemA[]
// ======

bits(8*size) MemA[bits(32) address, integer size]
    return MemA_with_priv[address, size, CurrentModeIsPrivileged()];

MemA[bits(32) address, integer size] = bits(8*size) value
    MemA_with_priv[address, size, CurrentModeIsPrivileged()] = value;
    return;

// MemA_unpriv[]
// =============

bits(8*size) MemA_unpriv[bits(32) address, integer size]
    return MemA_with_priv[address, size, FALSE];

MemA_unpriv[bits(32) address, integer size] = bits(8*size) value
    MemA_with_priv[address, size, FALSE] = value;
    return;

// MemA_with_priv[]
// ================

// Non-assignment form

bits(8*size) MemA_with_priv[bits(32) address, integer size, boolean privileged]

    // Sort out alignment
    if address == Align(address, size) then
        VA = address;
    elsif SCTLR.A == '1' || SCTLR.U == '1' then
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
            AlignmentFault(address, FALSE);
        else // if legacy non alignment-checking configuration
            VA = Align(address, size);

        // MMU or MPU
        memaddrdesc = TranslateAddress(VA, privileged, FALSE);

        // Memory array access, and sort out endianness
        value = _Mem[memaddrdesc, size];
        if CPSR.E == '1' then
            value = BigEndianReverse(value, size);

        return value;

    // Assignment form

    MemA_with_priv[bits(32) address, integer size, boolean privileged] = bits(8*size) value

        // Sort out alignment
        if address == Align(address, size) then
            VA = address;
        elsif SCTLR.A == '1' || SCTLR.U == '1' then
            AlignmentFault(address, FALSE);
        else // if legacy non alignment-checking configuration
            VA = Align(address, size);

        // MMU or MPU
        memaddrdesc = TranslateAddress(VA, privileged, TRUE);

        // Effect on exclusives
        if memaddrdesc.memattrs.shareable then
            ClearExclusiveByAddress(memaddrdesc.physicaladdress, ProcessorID(), size);

        // Sort out endianness, then memory array access
        if CPSR.E == '1' then
            value = BigEndianReverse(value, size);
        _Mem[memaddrdesc,size] = value;

        return;
```

B2.4.5 Unaligned memory accesses

The `MemU[]` function performs a memory access at the current privilege level, and the `MemU_unpriv[]` function performs an access that is always unprivileged.

In both cases:

- if the SCTLR.A bit is 0, unaligned accesses are supported
- if the SCTLR.A bit is 1, unaligned accesses produce Alignment faults.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
            AlignmentFault(address, FALSE);
        else // if legacy non alignment-checking configuration
            VA = Align(address, size);

        // MMU or MPU
        memaddrdesc = TranslateAddress(VA, privileged, FALSE);

        // Memory array access, and sort out endianness
        value = _Mem[memaddrdesc, size];
        if CPSR.E == '1' then
            value = BigEndianReverse(value, size);

        return value;

    // Assignment form

    MemA_with_priv[bits(32) address, integer size, boolean privileged] = bits(8*size) value

        // Sort out alignment
        if address == Align(address, size) then
            VA = address;
        elsif SCTLR.A == '1' || SCTLR.U == '1' then
            AlignmentFault(address, FALSE);
        else // if legacy non alignment-checking configuration
            VA = Align(address, size);

        // MMU or MPU
        memaddrdesc = TranslateAddress(VA, privileged, TRUE);

        // Effect on exclusives
        if memaddrdesc.memattrs.shareable then
            ClearExclusiveByAddress(memaddrdesc.physicaladdress, ProcessorID(), size);

        // Sort out endianness, then memory array access
        if CPSR.E == '1' then
            value = BigEndianReverse(value, size);
        _Mem[memaddrdesc,size] = value;

        return;
```


B2.4.5 アンアラインドメモリアクセス

`MemU[]` 関数は現在の特権レベルでメモリアクセスを実行しますが、`MemU_unpriv[]` 関数は常に非特権のアクセスを実行します。

いずれの場合も、次の規則が適用されます。

- SCTLR.A ビットが 0 の場合は、アンアラインドアクセスがサポートされます。
- SCTLR.A ビットが 1 の場合は、アンアラインドアクセスでアライメントフォールトが発生します。

注

ARMv7 以前のバージョンのアーキテクチャでは、SCTLR.A ビットと SCTLR.U ビットの両方が 0 の場合に、アンアラインドアクセスの下位アドレスビットが強制的に 0 に置き換えられてアラインされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

In versions of the architecture before ARMv7, if the SCTLR.A and SCTLR.U bits are both 0, an unaligned access is forced to be aligned by replacing the low-order address bits with zeros.

```
// MemU[]
// ======

bits(8*size) MemU[bits(32) address, integer size]
    return MemU_with_priv[address, size, CurrentModeIsPrivileged()];

MemU[bits(32) address, integer size] = bits(8*size) value
    MemU_with_priv[address, size, CurrentModeIsPrivileged()] = value;
    return;

// MemU_unpriv[]
// =============

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    return MemU_with_priv[address, size, FALSE];

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    MemU_with_priv[address, size, FALSE] = value;
    return;

// MemU_with_priv[]
// ================
//
// Due to single-copy atomicity constraints, the aligned accesses are distinguished from
// the unaligned accesses:
// * aligned accesses are performed at their size
// * unaligned accesses are expressed as a set of bytes.

// Non-assignment form

bits(8*size) MemU_with_priv[bits(32) address, integer size, boolean privileged]

    bits(8*size) value;

    // Legacy non alignment-checking configuration forces access to be aligned
    if SCTLR.A == '0' && SCTLR.U == '0' then address = Align(address, size);

    // Do aligned access, take alignment fault, or do sequence of bytes
    if address == Align(address, size) then
        value = MemA_with_priv[address, size, privileged];
    elsif SCTLR.A == '1' then
        AlignmentFault(address, FALSE);
    else // if unaligned access, SCTLR.A == '0', and SCTLR.U == '1'
        for i = 0 to size-1
            value<8*i+7:8*i> = MemA_with_priv[address+i, 1, privileged];
        if CPSR.E == '1' then
            value = BigEndianReverse(value, size);
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
// MemU[]
// ======

bits(8*size) MemU[bits(32) address, integer size]
    return MemU_with_priv[address, size, CurrentModeIsPrivileged()];

MemU[bits(32) address, integer size] = bits(8*size) value
    MemU_with_priv[address, size, CurrentModeIsPrivileged()] = value;
    return;

// MemU_unpriv[]
// =============

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    return MemU_with_priv[address, size, FALSE];

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    MemU_with_priv[address, size, FALSE] = value;
    return;

// MemU_with_priv[]
// ================
//
// Due to single-copy atomicity constraints, the aligned accesses are distinguished from
// the unaligned accesses:
// * aligned accesses are performed at their size
// * unaligned accesses are expressed as a set of bytes.

// Non-assignment form

bits(8*size) MemU_with_priv[bits(32) address, integer size, boolean privileged]

    bits(8*size) value;

    // Legacy non alignment-checking configuration forces access to be aligned
    if SCTLR.A == '0' && SCTLR.U == '0' then address = Align(address, size);

    // Do aligned access, take alignment fault, or do sequence of bytes
    if address == Align(address, size) then
        value = MemA_with_priv[address, size, privileged];
    elsif SCTLR.A == '1' then
        AlignmentFault(address, FALSE);
    else // if unaligned access, SCTLR.A == '0', and SCTLR.U == '1'
        for i = 0 to size-1
            value<8*i+7:8*i> = MemA_with_priv[address+i, 1, privileged];
        if CPSR.E == '1' then
            value = BigEndianReverse(value, size);

    return value;

// Assignment form

MemU_with_priv[bits(32) address, integer size, boolean privileged] = bits(8*size) value
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    return value;

// Assignment form

MemU_with_priv[bits(32) address, integer size, boolean privileged] = bits(8*size) value

    // Legacy non alignment-checking configuration forces access to be aligned
    if SCTLR.A == '0' && SCTLR.U == '0' then address = Align(address, size);

    // Do aligned access, take alignment fault, or do sequence of bytes
    if address == Align(address, size) then
        MemA_with_priv[address, value, privileged] = value;
    elsif SCTLR.A == '1' then
        AlignmentFault(address, TRUE);
    else // if unaligned access, SCTLR.A == '0', and SCTLR.U == '1'
        if CPSR.E == '1' then
            value = BigEndianReverse(value, size);
        for i = 0 to size-1
            MemA_with_priv[address+i, 1, privileged] = value<8*i+7:8*i>;

    return;
```

B2.4.6 Reverse endianness

The following pseudocode describes the operation to reverse endianness:

```
// BigEndianReverse()
// ==================

bits(8*N) BigEndianReverse (bits(8*N) value, integer N)
    assert N == 1 || N == 2 || N == 4 || N == 8;
    bits(8*N) result;
    case N of
        when 1
            result<7:0> = value<7:0>;
        when 2
            result<15:8> = value<7:0>;
            result<7:0> =  value<15:8>;
        when 4
            result<31:24> = value<7:0>;
            result<23:16> = value<15:8>;
            result<15:8> = value<23:16>;
            result<7:0> =  value<31:24>;
        when 8
            result<63:56> = value<7:0>;
            result<55:48> = value<15:8>
            result<47:40> = value<23:16>;
            result<39:32> = value<31:24>;
            result<31:24> = value<39:32>;
            result<23:16> = value<47:40>;
            result<15:8>  = value<55:48>;
            result<7:0>   = value<63:56>;
    return result;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    // Legacy non alignment-checking configuration forces access to be aligned
    if SCTLR.A == '0' && SCTLR.U == '0' then address = Align(address, size);

    // Do aligned access, take alignment fault, or do sequence of bytes
    if address == Align(address, size) then
        MemA_with_priv[address, value, privileged] = value;
    elsif SCTLR.A == '1' then
        AlignmentFault(address, TRUE);
    else // if unaligned access, SCTLR.A == '0', and SCTLR.U == '1'
        if CPSR.E == '1' then
            value = BigEndianReverse(value, size);
        for i = 0 to size-1
            MemA_with_priv[address+i, 1, privileged] = value<8*i+7:8*i>;

    return;
```

B2.4.6 エンディアン形式の反転

次の擬似コードは、エンディアン形式を反転するための操作を示しています。

```
// BigEndianReverse()
// ==================

bits(8*N) BigEndianReverse (bits(8*N) value, integer N)
    assert N == 1 || N == 2 || N == 4 || N == 8;
    bits(8*N) result;
    case N of
        when 1
            result<7:0> = value<7:0>;
        when 2
            result<15:8> = value<7:0>;
            result<7:0> =  value<15:8>;
        when 4
            result<31:24> = value<7:0>;
            result<23:16> = value<15:8>;
            result<15:8> = value<23:16>;
            result<7:0> =  value<31:24>;
        when 8
            result<63:56> = value<7:0>;
            result<55:48> = value<15:8>
            result<47:40> = value<23:16>;
            result<39:32> = value<31:24>;
            result<31:24> = value<39:32>;
            result<23:16> = value<47:40>;
            result<15:8>  = value<55:48>;
            result<7:0>   = value<63:56>;
    return result;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.4.7 Exclusive monitors operations

The SetExclusiveMonitors() function sets the exclusive monitors for a Load-Exclusive instruction. The ExclusiveMonitorsPass() function checks whether a Store-Exclusive instruction still has possession of the exclusive monitors and therefore completes successfully.

```
// SetExclusiveMonitors()
// ======================

SetExclusiveMonitors(bits(32) address, integer size)

    memaddrdesc = TranslateAddress(address, CurrentModeIsPrivileged(), FALSE);

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.physicaladdress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.physicaladdress, ProcessorID(), size);

// ExclusiveMonitorsPass()
// =======================

boolean ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusive Monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    if address != Align(address, size) then
        AlignmentFault(address, TRUE);
    else
        memaddrdesc = TranslateAddress(address, CurrentModeIsPrivileged(), TRUE);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    if memaddrdesc.memattrs.shareable then
        passed = passed && IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    if passed then
        ClearExclusiveLocal(ProcessorID());

    return passed;
```

The MarkExclusiveGlobal() procedure takes as arguments a FullAddress paddress, the processor identifier processorid and the size of the transfer. The procedure records that processor processorid has requested exclusive access covering at least size bytes from address paddress. The size of region marked as exclusive is IMPLEMENTATION DEFINED, up to a limit of 2KB, and no smaller than two words, and aligned in the address space to the size of the region. It is UNPREDICTABLE whether this causes any previous request for exclusive access to any other address by the same processor to be cleared.

```
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size)
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.4.7 排他モニタ操作

SetExclusiveMonitors() 関数は、排他ロード命令用に排他モニタを設定します。ExclusiveMonitorsPass() 関数は、排他ストア命令が排他モニタを所有しており、正常に終了するかどうかをチェックします。

```
// SetExclusiveMonitors()
// ======================

SetExclusiveMonitors(bits(32) address, integer size)

    memaddrdesc = TranslateAddress(address, CurrentModeIsPrivileged(), FALSE);

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.physicaladdress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.physicaladdress, ProcessorID(), size);

// ExclusiveMonitorsPass()
// =======================

boolean ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusive Monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    if address != Align(address, size) then
        AlignmentFault(address, TRUE);
    else
        memaddrdesc = TranslateAddress(address, CurrentModeIsPrivileged(), TRUE);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    if memaddrdesc.memattrs.shareable then
        passed = passed && IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    if passed then
        ClearExclusiveLocal(ProcessorID());

    return passed;
```

MarkExclusiveGlobal() プロシージャは、FullAddress の paddress、プロセッサ識別子の processorid、および転送サイズを引数として受け取ります。このプロシージャは、プロセッサ processorid が、アドレス paddress から少なくとも size バイトをカバーする排他的アクセスを要求したことを記録します。排他的としてマークされる領域のサイズは実装定義で、2 ワード以上 2KB 以下に制限され、アドレス空間で領域のサイズにアラインされます。これによって、過去に同じプロセッサが他のアドレスに対して要求した排他的アクセスがクリアされるかどうかは、予測不能です。

```
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size)
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The `MarkExclusiveLocal()` procedure takes as arguments a FullAddress `paddress`, the processor identifier `processorid` and the size of the transfer. The procedure records in a local record that processor `processorid` has requested exclusive access to an address covering at least `size` bytes from address paddress. The size of the region marked as exclusive is IMPLEMENTATION DEFINED, and can at its largest cover the whole of memory, but is no smaller than two words, and is aligned in the address space to the size of the region. It is IMPLEMENTATION DEFINED whether this procedure also performs a `MarkExclusiveGlobal()` using the same parameters.

```
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size)
```

The `IsExclusiveGlobal()` function takes as arguments a FullAddress `paddress`, the processor identifier `processorid` and the size of the transfer. The function returns TRUE if the processor `processorid` has marked in a global record an address range as exclusive access requested that covers at least the `size` bytes from address `paddress`. It is IMPLEMENTATION DEFINED whether it returns TRUE or FALSE if a global record has marked a different address as exclusive access requested. If no address is marked in a global record as exclusive access, `IsExclusiveGlobal()` returns FALSE.

```
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size)
```

The `IsExclusiveLocal()` function takes as arguments a FullAddress `paddress`, the processor identifier `processorid` and the size of the transfer. The function returns TRUE if the processor `processorid` has marked an address range as exclusive access requested that covers at least the `size` bytes from address `paddress`. It is IMPLEMENTATION DEFINED whether this function returns TRUE or FALSE if the address marked as exclusive access requested does not cover all of the `size` bytes from address `paddress`. If no address is marked as exclusive access requested, then this function returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of `IsExclusiveGlobal()` with the same parameters.

```
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size)
```

The `ClearExclusiveByAddress()` procedure takes as arguments a FullAddress `paddress`, the processor identifier `processorid` and the size of the transfer. The procedure clears the global records of all processors, other than `processorid`, for which an address region including any of the `size` bytes starting from `paddress` has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether the equivalent global record of the processor `processorid` is also cleared if any of the `size` bytes starting from `paddress` has had a request for an exclusive access, or if any other address has had a request for an exclusive access.

```
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size)
```

The `ClearExclusiveLocal()` procedure takes as arguments the processor identifier `processorid`. The procedure clears the local record of processor `processorid` for which an address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of processor `processorid` that an address has had a request for an exclusive access.

```
ClearExclusiveLocal(integer processorid)
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`MarkExclusiveLocal()` プロシージャは、FullAddress の `paddress`、プロセッサ識別子の `processorid`、および転送サイズを引数として受け取ります。このプロシージャは、プロセッサ `processorid` が、アドレス paddress から少なくとも `size` バイトをカバーするアドレスに対して排他的アクセスを要求したことを、ローカルレコードに記録します。排他的としてマークされる領域のサイズは実装定義で、最大でメモリ全体をカバーすることができますが、最低 2 ワードで、アドレス空間内で領域のサイズにアラインされます。このプロシージャが同じパラメータを使用して `MarkExclusiveGlobal()` も実行するかどうかは、実装定義です。

```
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size)
```

`IsExclusiveGlobal()` 関数は、FullAddress の `paddress`、プロセッサ識別子の `processorid`、および転送サイズを引数として受け取ります。プロセッサ `processorid` が、アドレス `paddress` から少なくとも `size` バイトをカバーするアドレス範囲を、排他的アクセス用に要求したことがグローバルレコードにマークされている場合、この関数は TRUE を返します。グローバルレコードで、別のアドレスが排他的アクセスとして要求されたことがマークされている場合、この関数が TRUE と FALSE のどちらを返すかは実装定義です。排他的アクセスとしてマークされたアドレスがグローバルレコード内に存在しない場合、`IsExclusiveGlobal()` は FALSE を返します。

```
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size)
```

`IsExclusiveLocal()` 関数は、FullAddress の `paddress`、プロセッサ識別子の `processorid`、および転送サイズを引数として受け取ります。プロセッサ `processorid` が、アドレス `paddress` から少なくとも `size` バイトをカバーするアドレス範囲を、排他的アクセス用に要求したことがマークされている場合、この関数は TRUE を返します。排他的アクセスの要求としてマークされたアドレスが、アドレス `paddress` から `size` バイトの全体をカバーしていない場合、この関数が TRUE と FALSE のどちらを返すかは実装定義です。排他的アクセスの要求としてマークされたアドレスが存在しない場合、この関数は FALSE を返します。この関数の結果と、同じパラメータを使用した `IsExclusiveGlobal()` の結果との間で論理積が実行されるかどうかは、実装定義です。

```
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size)
```

`ClearExclusiveByAddress()` プロシージャは、FullAddress の `paddress`、プロセッサ識別子の `processorid`、および転送サイズを引数として受け取ります。このプロシージャは、グローバルレコードのうち、`paddress` から始まる `size` バイトのいずれかを含むアドレス領域について排他的アクセスを要求しているものを、`processorid` のプロセッサでなければすべてクリアします。プロセッサ `processorid` についても、同様に `paddress` から始まる `size` バイトのいずれかに排他的アクセスを要求している、または他のアドレスに排他的アクセスを要求しているグローバルレコードをクリアするかどうかは、実装定義です。

```
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size)
```

`ClearExclusiveLocal()` プロシージャは、プロセッサ識別子の `processorid` を引数として受け取ります。このプロシージャは、プロセッサ `processorid` のローカルレコードで、あるアドレスに排他的アクセスを要求しているものをクリアします。この操作で、プロセッサ `processorid` のグローバルレコードについても、アドレスに排他的アクセスを要求している場合にクリアするかどうかは、実装定義です。

```
ClearExclusiveLocal(integer processorid)
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.4.8 Access permission checking

The function `CheckPermission()` is used by both the VMSA and PMSA architectures to perform access permission checking based on attributes derived from the translation tables or region descriptors. The `domain` and `sectionnotpage` arguments are only relevant for the VMSA architecture.

The interpretation of the access permissions is shown in:

- *Access permissions* on page B3-28, for a VMSA implementation
- *Access permissions* on page B4-9, for a PMSA implementation.

The following pseudocode describes the checking of the access permission:

```
// CheckPermission()
// ================

CheckPermission(Permissions perms, bits(32) mva,
                boolean sectionnotpage, bits(4) domain, boolean iswrite, boolean ispriv)

    if SCTLR.AFE == '0' then
        perms.ap<0> = '1';

    case perms.ap of
        when '000'  abort = TRUE;
        when '001'  abort = !ispriv;
        when '010'  abort = !ispriv && iswrite;
        when '011'  abort = FALSE;
        when '100'  UNPREDICTABLE;
        when '101'  abort = !ispriv || iswrite;
        when '110'  abort = iswrite;
        when '111'
            if MemorySystemArchitecture() == MemArch_VMSA then
                abort = iswrite
            else
                UNPREDICTABLE;

   if abort then
       DataAbort(mva, domain, sectionnotpage, iswrite, DAbort_Permission);

   return;
```

B2.4.9 Default memory access decode

The function `DefaultTEXDecode()` is used by both the VMSA and PMSA architectures to decode the `texcb` and `S` attributes derived from the translation tables or region descriptors.

The interpretation of the arguments is shown in:

- *C, B, and TEX[2:0] encodings without TEX remap* on page B3-33, for a VMSA implementation
- *C, B, and TEX[2:0] encodings* on page B4-11, for a PMSA implementation.

The following pseudocode describes the default memory access decoding, when memory region remapping is not implemented:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.4.8 アクセス許可のチェック

`CheckPermission()` 関数は、変換テーブルまたは領域記述子から抽出された属性に基づくアクセス許可チェックを実行するため、VMSA と PMSA の両方のアーキテクチャで使用されます。VMSA アーキテクチャの場合は、`domain` 引数と `sectionnotpage` 引数だけが関係します。

アクセス許可の解釈については、次のセクションを参照して下さい。

- *アクセス許可* P. B3-28（VMSA 実装の場合）
- *アクセス許可* P. B4-9（PMSA 実装の場合）

次の擬似コードは、アクセス許可のチェック方法を示しています。

```
// CheckPermission()
// ================

CheckPermission(Permissions perms, bits(32) mva,
                boolean sectionnotpage, bits(4) domain, boolean iswrite, boolean ispriv)

    if SCTLR.AFE == '0' then
        perms.ap<0> = '1';

    case perms.ap of
        when '000'  abort = TRUE;
        when '001'  abort = !ispriv;
        when '010'  abort = !ispriv && iswrite;
        when '011'  abort = FALSE;
        when '100'  UNPREDICTABLE;
        when '101'  abort = !ispriv || iswrite;
        when '110'  abort = iswrite;
        when '111'
            if MemorySystemArchitecture() == MemArch_VMSA then
                abort = iswrite
            else
                UNPREDICTABLE;

   if abort then
       DataAbort(mva, domain, sectionnotpage, iswrite, DAbort_Permission);

   return;
```

B2.4.9 デフォルトのメモリアクセス デコード

`DefaultTEXDecode()` 関数は、変換テーブルまたは領域記述子から抽出された `texcb` 属性と `S` 属性をデコードするために、VMSA と PMSA の両方のアーキテクチャで使用されます。

引数の解釈については、次のセクションを参照して下さい。

- *TEX 再マップを伴わない C、B、TEX[2:0] のエンコード* P. B3-33（VMSA 実装の場合）
- *C、B、TEX[2:0] のエンコード* P. B4-11（PMSA 実装の場合）

次の擬似コードは、メモリ領域の再マッピングが実装されていない場合の、デフォルトのメモリアクセス デコードを示しています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
// DefaultTEXDecode()
// ==================

MemoryAttributes DefaultTEXDecode(bits(5) texcb, bit S)

    MemoryAttributes memattrs;

    case texcb of
        when '00000'
            memattrs.type = MemType_StronglyOrdered;
            memattrs.innerattrs = '00';  // Non-cacheable
            memattrs.outerattrs = '00';  // Non-cacheable
            memattrs.shareable = TRUE;
        when '00001'
            memattrs.type = MemType_Device;
            memattrs.innerattrs = '00';  // Non-cacheable
            memattrs.outerattrs = '00';  // Non-cacheable
            memattrs.shareable = TRUE;
        when '0001x', '00100'
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = texcb<1:0>;
            memattrs.outerattrs = texcb<1:0>;
            memattrs.shareable = (S == '1');
        when '00110'
            IMPLEMENTATION_DEFINED setting of memattrs;
        when '00111'
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = '01';  // Write-back write-allocate cacheable
            memattrs.outerattrs = '01';  // Write-back write-allocate cacheable
            memattrs.shareable = (S == '1');
        when '01000'
            memattrs.type = MemType_Device;
            memattrs.innerattrs = '00';  // Non-cacheable
            memattrs.outerattrs = '00';  // Non-cacheable
            memattrs.shareable = FALSE;
        when '1xxxx'
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = texcb<1:0>;
            memattrs.outerattrs = texcb<3:2>;
            memattrs.shareable = (S == '1');
        otherwise
            UNPREDICTABLE;

    memattrs.outershareable = memattrs.shareable;

    return memattrs;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳


```
// DefaultTEXDecode()
// ==================

MemoryAttributes DefaultTEXDecode(bits(5) texcb, bit S)

    MemoryAttributes memattrs;

    case texcb of
        when '00000'
            memattrs.type = MemType_StronglyOrdered;
            memattrs.innerattrs = '00';  // Non-cacheable
            memattrs.outerattrs = '00';  // Non-cacheable
            memattrs.shareable = TRUE;
        when '00001'
            memattrs.type = MemType_Device;
            memattrs.innerattrs = '00';  // Non-cacheable
            memattrs.outerattrs = '00';  // Non-cacheable
            memattrs.shareable = TRUE;
        when '0001x', '00100'
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = texcb<1:0>;
            memattrs.outerattrs = texcb<1:0>;
            memattrs.shareable = (S == '1');
        when '00110'
            IMPLEMENTATION_DEFINED setting of memattrs;
        when '00111'
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = '01';  // Write-back write-allocate cacheable
            memattrs.outerattrs = '01';  // Write-back write-allocate cacheable
            memattrs.shareable = (S == '1');
        when '01000'
            memattrs.type = MemType_Device;
            memattrs.innerattrs = '00';  // Non-cacheable
            memattrs.outerattrs = '00';  // Non-cacheable
            memattrs.shareable = FALSE;
        when '1xxxx'
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = texcb<1:0>;
            memattrs.outerattrs = texcb<3:2>;
            memattrs.shareable = (S == '1');
        otherwise
            UNPREDICTABLE;

    memattrs.outershareable = memattrs.shareable;

    return memattrs;
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.4.10 Data Abort exception

The `DataAbort()` function generates a Data Abort exception and is used by both the VMSA and PMSA architectures. It sets the DFSR to indicate:

- the type of the abort, including the distinction between section and page on a VMSA implementation
- on a VMSA implementation, the domain, if appropriate
- whether the access was a read or write.

For a synchronous abort it also sets the DFAR to the MVA of the abort.

For details of the FSR encoding values see:

- *Fault Status Register encodings for the VMSA* on page B3-50, for a VMSA implementation
- *Fault Status Register encodings for the PMSA* on page B4-19, for a PMSA implementation.

An implementation might also set the IMPLEMENTATION DEFINED ADFSR.

```
// Data abort types.

enumeration DAbort {DAbort_AccessFlag,
                    DAbort_Alignment,
                    DAbort_Background,
                    DAbort_Domain,
                    DAbort_Permission,
                    DAbort_Translation};

DataAbort(bits(32) address, bits(4) domain, boolean sectionnotpage, boolean iswrite, DAbort type)
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B2.4.10 データアボート例外

`DataAbort()` 関数はデータアボート例外を生成するもので、VMSA と PMSA の両方のアーキテクチャで使用されます。この関数は、DFSR の設定により次の情報を示します。

- アボートのタイプ。これには、VMSA 実装でのセクションとページの区別が含まれます。
- VMSA 実装の場合、ドメイン（該当する場合）。
- アクセスが読み出しか、書き込みか。

同期アボートの場合、DFAR がアボートの MVA に設定されます。

FSR エンコードの値の詳細については、次のセクションを参照して下さい。

- *VMSA のフォールト ステータス レジスタのエンコード* P. B3-50（VMSA 実装の場合）
- *PMSA のフォールト ステータス レジスタのエンコード* P. B4-19（PMSA 実装の場合）

実装で、実装定義の ADFSR を設定することもできます。


```
// Data abort types.

enumeration DAbort {DAbort_AccessFlag,
                    DAbort_Alignment,
                    DAbort_Background,
                    DAbort_Domain,
                    DAbort_Permission,
                    DAbort_Translation};

DataAbort(bits(32) address, bits(4) domain, boolean sectionnotpage, boolean iswrite, DAbort type)
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter B3
Virtual Memory System Architecture (VMSA)

This chapter provides a system-level view of the Virtual Memory System Architecture (VMSA), the memory system architecture of an ARMv7-A implementation. It contains the following sections:

- *About the VMSA* on page B3-2
- *Memory access sequence* on page B3-4
- *Translation tables* on page B3-7
- *Address mapping restrictions* on page B3-23
- *Secure and Non-secure address spaces* on page B3-26
- *Memory access control* on page B3-28
- *Memory region attributes* on page B3-32
- *VMSA memory aborts* on page B3-40
- *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48
- *Translation Lookaside Buffers (TLBs)* on page B3-54
- *Virtual Address to Physical Address translation operations* on page B3-63
- *CP15 registers for a VMSA implementation* on page B3-64
- *Pseudocode details of VMSA memory system operations* on page B3-156.

Note

For an ARMv7-A implementation, this chapter must be read with Chapter B2 *Common Memory System Architecture Features*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 B3 章
仮想メモリシステム アーキテクチャ (VMSA)

本章では、ARMv7-A 実装のメモリシステム アーキテクチャである仮想メモリシステム アーキテクチャ (VMSA) の、システムレベルのビューについて説明します。本章は次のセクションから構成されています。

- *VMSA について* P. B3-2
- *メモリアクセス シーケンス* P. B3-4
- *変換テーブル* P. B3-7
- *アドレスマッピングの制限* P. B3-23
- *セキュアアドレス空間と非セキュアアドレス空間* P. B3-26
- *メモリアクセス制御* P. B3-28
- *メモリ領域の属性* P. B3-32
- *VMSA メモリアボート* P. B3-40
- *VMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B3-48
- *変換ルックアサイドバッファ (TLB)* P. B3-54
- *仮想アドレスから物理アドレスへの変換操作* P. B3-63
- *VMSA 実装の CP15 レジスタ* P. B3-64
- *VMSA メモリシステム操作の擬似コードの詳細* P. B3-156

注

ARMv7-A 実装については、本章と第 B2 章 *「一般的なメモリシステム アーキテクチャの機能」* をあわせて読んで下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.1 About the VMSA

Complex operating systems typically use a virtual memory system to provide separate, protected address spaces for different processes. The ARMv7 VMSA is referred to as VMSAv7. For details of the differences in previous versions of the ARM architecture see:

- *Virtual memory support* on page AppxH-21 for the ARMv4 and ARMv5 architectures
- *Virtual memory support* on page AppxG-24 for ARMv6.

In a VMSA, a *Memory Management Unit* (MMU) provides facilities that enable an operating system to dynamically allocate memory and other memory-mapped system resources to the processes. The MMU provides fine-grained control of a memory system through a set of virtual to physical address mappings and associated memory properties held in memory-mapped tables known as translation tables.

The translation properties associated with each translation table entry include:

Memory access permission control

This controls whether a program has access to a memory area. The possible settings are no access, read-only access, or read/write access. In addition, there is control of whether code can be executed from the memory area.

If a processor attempts an access that is not permitted, a memory abort is signaled to the processor.

The permitted level of access can be affected by:

- whether the program is running in User mode or a privileged mode
- the use of domains.

Memory region attributes

These describe the properties of a memory region. The top-level attribute, the Memory type, is one of Strongly-ordered, Device, or Normal. Device and Normal memory regions have additional attributes, see *Summary of ARMv7 memory attributes* on page A3-25.

Virtual-to-physical address mapping

The VMSA regards the address of an explicit data access or an instruction fetch as a *Virtual Address* (VA). The MMU maps this address onto the required *Physical Address* (PA).

VA to PA address mapping can be used to manage the allocation of physical memory in many ways. For example:

- to allocate memory to different processes with potentially conflicting address maps
- to enable an application with a sparse address map to use a contiguous region of physical memory.

A full translation table lookup is called a *translation table walk*. It is performed automatically by hardware, and has a significant cost in execution time, requiring at least one main memory access, and often two. *Translation Lookaside Buffers* (TLBs) reduce the average cost of a memory access by caching the results of translation table walks. TLBs behave as caches of the translation table information, and the VMSA provides TLB maintenance operations to manage TLB contents in software.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.1 VMSA について

複雑なオペレーティングシステムでは一般に、プロセスごとに独立の保護されたアドレス空間を提供するため、仮想メモリシステムが使用されます。ここでは、ARMv7 VMSA を VMSAv7 と表記します。以前のバージョンの ARM アーキテクチャと比較した相違点の詳細については、次のセクションを参照して下さい。

- *仮想メモリのサポート* P. 付録 H-21（ARMv4 と ARMv5 アーキテクチャの場合）
- *仮想メモリのサポート* P. 付録 G-24 （ARMv6 の場合）

VMSA では、*メモリ管理ユニット*(MMU) を通して、オペレーティングシステムがメモリや他のメモリマップされたシステムリソースを、動的にプロセスへ割り当てる仕組みが提供されます。MMU によって、変換テーブルと呼ばれるメモリマップされたテーブルに保持されている、仮想アドレスから物理アドレスへのマッピングや関連するメモリプロパティのセットにより、メモリシステムを細かく制御できます。

各変換テーブルエントリに関連付けられた変換プロパティには、次のものが含まれます。

メモリへのアクセス許可の制御

プログラムがメモリ領域にアクセスできるかどうかの制御に使用されます。アクセス不可、読み出し専用アクセス、読み出し / 書き込みアクセスのいずれかに設定することができます。その他に、メモリ領域からコードを実行できるかどうかの制御も含まれます。

プロセッサが許可されていないアクセスを試みた場合は、そのプロセッサにメモリアボートが通知されます。

許可されるアクセスレベルは、次の要因の影響を受ける可能性があります。

- プログラムがユーザモードと特権モードのどちらで実行しているか
- ドメインの使用

メモリ領域属性

メモリ領域のプロパティを示します。最も上位の属性はメモリタイプで、ストロングリオーダ、デバイス、ノーマルのいずれかです。デバイスとノーマルのメモリ領域には、他の属性が付与されます。*ARMv7 のメモリ属性の要約* P. A3-25 を参照して下さい。

仮想アドレスから物理アドレスへのマッピング

VMSA では、明示的なデータアクセスまたは命令フェッチのアドレスは、*仮想アドレス*(VA) と見なされます。このアドレスは MMU により、必要な*物理アドレス*(PA) にマッピングされます。

VA から PA へのマッピングを使用して、各種の方法で物理メモリの割り当てを管理できます。次に例を示します。

- アドレスマップが競合している可能性のある複数のプロセスにメモリを割り当てる。
- アドレスマップを不連続に使用しているアプリケーションで、連続する物理メモリ領域を使用可能にする。

変換テーブル全体のルックアップは、*変換テーブルウォーク*と呼ばれます。これは、ハードウェアによって自動的に実行されますが、少なくとも 1 回、多くの場合は 2 回のメインメモリアクセスを必要とする実行時間の長い処理です。*変換ルックアサイドバッファ*(TLB) を使用す

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

To reduce the software overhead of TLB maintenance, the VMSA distinguishes between *Global pages* and *Process specific pages*. The *Address Space Identifier* (ASID) identifies pages associated with a specific process and provides a mechanism for changing process specific tables without having to perform maintenance on the TLB structures.

System Control coprocessor (CP15) registers control the VMSA, including defining the location of the translation tables. They include registers that contain memory fault status and address information. See *CP15 registers for a VMSA implementation* on page B3-64. When the Security Extensions are implemented, many of the CP15 registers are banked between the Secure and Non-secure security states. This means separate system control software can be used in the different security states.

VMSAv7 supports physical addresses of up to 40 bits, though implementations are permitted to support only 32 bits of physical address. Where implementations support more than 32 bits of physical address, generating physical addresses with PA[39:32] != 0b00000000 requires the use of *Supersections*, see *Translation tables* on page B3-7.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

れば、変換テーブルウォークの結果をキャッシュすることによって、平均メモリアクセス時間が短縮されます。TLB は変換テーブル情報のキャッシュとして機能します。VMSA では、TLB の内容をソフトウェアで管理するための TLB メンテナンス操作が提供されます。

TLB 保守のソフトウェアオーバヘッドを低減するため、VMSA では*グローバルページ*と*プロセス固有のページ*が区別されます。*アドレス空間識別子*(ASID) によって、特定のプロセスに関連付けられたページが識別され、TLB 構造の保守を行わずにプロセス固有のテーブルを変更可能な機構が提供されます。

VMSA は、システム制御コプロセッサ (CP15) レジスタを使用して制御します。これには、変換テーブルの場所の定義も含まれます。CP15 レジスタには、メモリフォールト ステータスやアドレス情報を保持するレジスタが含まれます。詳細については、*VMSA 実装の CP15 レジスタ* P. B3-64 を参照して下さい。セキュリティ拡張機能が実装されている場合は、CP15 レジスタの多くが、セキュアと非セキュアのセキュリティ状態の間でバンクされます。これは、セキュリティ状態ごとに異なるシステム制御ソフトウェアを使用可能なことを意味します。

VMSAv7 では最大 40 ビットの物理アドレスがサポートされますが、実装では 32 ビットの物理アドレスのサポートしか許可されません。実装で 32 ビットを超える物理アドレスをサポートする場合は、PA[39:32] != 0b00000000 の物理アドレスを生成するために*スーパーセクション*を使用する必要があります。*変換テーブル* P. B3-7 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.2 Memory access sequence

Explicit data accesses and instruction fetches generate memory accesses using VAs. The VA for an access is subject to two stages of address translation:

1. A translation from VA to *Modified Virtual Address* (MVA) by the FCSE, if it is implemented, see *FCSE translation*
2. A translation from MVA to PA using the translation tables, see *Translation from MVA to PA using the translation tables*.

B3.2.1 FCSE translation

The FCSE translation is a linear remapping of the bottom 32MBytes of the Virtual Address map, to a 32MByte address block determined by the FCSEIDR, see *c13, FCSE Process ID Register (FCSEIDR)* on page B3-152. Therefore, the translation is that shown by the pseudo-function `FCSETranslate`, see *FCSE translation* on page B3-156.

Note

- The FCSE translation has no effect if bits FCSEIDR[31:25] are 0b0000000, see *c13, FCSE Process ID Register (FCSEIDR)* on page B3-152.
- From VMSAv6, use of FCSE translation is deprecated.
- In VMSAv7, the FCSE is optional and might not be implemented. If it is not implemented the VMSA behavior is that MVA = VA, and the FCSEIDR register is RAZ/WI.

B3.2.2 Translation from MVA to PA using the translation tables

The MMU translates the MVA to the PA. Typically, this translation attempts to find the translation table entry held in a TLB that either:

- is a global entry
- was brought into the TLB with the ASID that matches the current value held in the CONTEXTIDR, see *c13, Context ID Register (CONTEXTIDR)* on page B3-153.

If no matching entry is found in a TLB, then the hardware locates the appropriate entry in the translation tables held in memory.

When the translation table entry is located, in the TLB or in memory, either:

- It is not a valid translation, and therefore causes a Translation fault.
- The contents of the entry contain the PA, the memory permission attributes and the memory type attributes for the required access. Using the entry might cause an abort, for variety of reasons.

For more information about MMU faults see *MMU faults* on page B3-40.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.2 メモリアクセス シーケンス

明示的なデータアクセスと命令フェッチによって、VA を使用したメモリアクセスが生成されます。アクセスの VA には、次の 2 段階のアドレス変換が行われます。

1. FCSE（実装されている場合）による VA から*修飾仮想アドレス*(MVA) への変換（*FCSE 変換*を参照）
2. 変換テーブルを使用した MVA から PA への変換（*変換テーブルを使用した MVA から PA への変換*を参照）

B3.2.1 FCSE 変換

FCSE 変換は、仮想アドレスマップの下位 32MB から、FCSEIDR によって決定される 32MB のアドレスブロックへの線形再マッピングです。*c13、FCSE プロセス ID レジスタ (FCSEIDR)* P. B3- 152 を参照して下さい。したがって、この変換は、擬似関数 `FCSETranslate` によって表現される変換です。*FCSE 変換* P. B3- 156 を参照して下さい。

注

- ビット FCSEIDR[31:25] が 0b0000000 の場合、FCSE 変換は無効です。*c13、FCSE プロセス ID レジスタ (FCSEIDR)* P. B3- 152 を参照して下さい。
- VMSAv6 およびそれ以降では、FCSE 変換の使用は非推奨です。
- VMSAv7 では FCSE はオプションで、実装されていない可能性があります。FCSE が実装されていない場合、VMSA は MVA = VA として動作し、FCSEIDR レジスタは RAZ/WI です。

B3.2.2 変換テーブルを使用した MVA から PA への変換

MMU は、MVA を PA に変換します。通常は、この変換によって、次のどちらかの TLB 内に保持された変換テーブルエントリが検索されます。

- グローバルエントリ
- CONTEXTIDR に保持されている現在の値と一致する ASID を持つ TLB（*c13、コンテキスト ID レジスタ (CONTEXTIDR)* P. B3- 153 を参照）

一致するエントリが TLB に存在しない場合は、ハードウェアによって、メモリに保持された変換テーブル内の該当するエントリが検索されます。

変換テーブルエントリが TLB またはメモリで特定されると、次のいずれかの動作が行われます。

- 有効な変換でないため、変換フォールトが発生する。
- エントリの内容に、必要なアクセスの PA、メモリ許可属性、およびメモリタイプ属性が含まれている。このエントリを使用すると、さまざまな理由から、アボートが発生する可能性があります。

MMU フォールトの詳細については、*MMU フォールト* P. B3- 40 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.2.3 Enabling and disabling the MMU

The MMU can be enabled and disabled by writing to the SCTLR.M bit, see *c1, System Control Register (SCTLR)* on page B3-96. On reset, this bit is cleared to 0, disabling the MMU.

When the MMU is disabled, memory accesses are treated as follows:

- All data accesses are treated as Non-cacheable and Strongly-ordered. Unexpected data cache hit behavior is IMPLEMENTATION DEFINED.
- The treatment of instruction accesses depends on the value of the SCTLR.I bit:

  When I == 0

  All instruction accesses are Non-cacheable.

  When I == 1

  All instruction accesses are Cacheable:
  - Inner Write-Through no Write-Allocate
  - Outer Write-Through no Write-Allocate.

  In both cases all instruction accesses are Non-shareable, Normal memory.

  Note

  On some implementations, if the SCTLR.TRE bit is set to 0 then this behavior can be changed by the remap settings in the memory remap registers, see *CP15 c10, Memory Remap Registers* on page B3-143. The details of TEX remapping when SCTLR.TRE is set to 0 are IMPLEMENTATION DEFINED, see *SCTLR.TRE, SCTLR.M, and the effect of the MMU remap registers* on page B3-38.
- No memory access permission checks are performed, and no aborts are generated by the MMU.
- For every access the PA is equal to the MVA. This is known as a flat address mapping.
- If the FCSE is implemented, the FCSE PID is SBZ when the MMU is disabled. This is the reset value for the FCSE PID. Behavior is UNPREDICTABLE if the FCSE PID is not zero when the MMU is disabled.

  When the FCSE is implemented software must clear the FCSE PID before disabling the MMU.
- CP15 cache maintenance operations act on the target cache whether the MMU is enabled or not, and regardless of the values of the memory attributes. However, if the MMU is disabled, they use the flat address mapping, and all mappings are considered global.

  CP15 TLB invalidate operations act on the target TLB whether the MMU is enabled or not.

All relevant CP15 registers must be programmed before the MMU is enabled. This includes setting up suitable translation tables in memory.

When the MMU is disabled, an instruction can be fetched if one of the following conditions is met:

- The instruction is in the same 4KB block of memory (aligned to 4KB) as an instruction that is required by a simple sequential execution of the program, or is in the 4KB block of memory immediately following such a block.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.2.3 MMU の許可と禁止

MMU は、SCTLR.M ビットへの書き込みにより許可または禁止できます。*c1、システム制御レジスタ (SCTLR)* P. B3-96 を参照して下さい。このビットはリセット時に 0 にクリアされ、MMU が非稼動になります。

MMU が非稼動の場合、メモリアクセスは次のように処理されます。

- すべてのデータアクセスは、キャッシュ不可かつストロングリオーダとして扱われます。予期しないデータキャッシュ ヒットの動作は実装定義です。
- 命令アクセスの扱いは、次のように SCTLR.I ビットの値によって異なります。

  I = 0 の場合

  すべての命令アクセスはキャッシュ不可です。

  I = 1 の場合

  すべての命令アクセスはキャッシュ可能で、次の規則に従います。
  - 内部ライトスルー、書き込み割り当てなし
  - 外部ライトスルー、書き込み割り当てなし

  どちらの場合も、すべての命令アクセスは共有不可、ノーマルメモリです。

  注

  一部の実装では、SCTLR.TRE ビットが 0 にセットされている場合に、メモリ再マップレジスタ内の再マップ設定によってこの動作を変更することができます。*CP15 c10、メモリ再マップレジスタ* P. B3-143 を参照して下さい。SCTLR.TRE が 0 にセットされている場合の TEX 再マッピングの詳細は、実装定義です。*SCTLR.TRE、SCTLR.M、および MMU 再マップレジスタの効果* P. B3-38 を参照して下さい。
- メモリアクセス許可のチェックは実行されず、MMU によるアボートは生成されません。
- すべてのアクセスで PA が MVA と一致します。これをフラット アドレスマッピングと呼びます。
- FCSE が実装されており、MMU が非稼動の場合は、FCSE PID は SBZ です。これは FCSE PID のリセット時の値です。MMU が非稼動で、FCSE PID が 0 でない場合の動作は予測不能です。

  FCSE が実装されている場合は、MMU を非稼動にする前にソフトウェアで FCSE PID をクリアする必要があります。
- CP15 キャッシュ保守操作は、MMU が稼動しているかどうか、およびメモリ属性の値にかかわらず、ターゲットのキャッシュに作用します。ただし、MMU が非稼動の場合はフラット アドレスマッピングが使用され、すべてのマッピングがグローバルと見なされます。

  CP15 TLB 無効化操作は、MMU が稼動しているかどうかにかかわらず、ターゲットの TLB に作用します。

関連するすべての CP15 レジスタは、MMU を稼動する前にプログラムする必要があります。これには、メモリ内の適切な変換テーブルのセットアップが含まれます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- The instruction is in the same 4KB block of memory (aligned to 4KB) from which an instruction has previously been required by a simple sequential execution of the program with the MMU disabled, or is in the 4KB block immediately following such a block.

Note

- Software must ensure that instructions that will be executed when the MMU is disabled are located within 4KB blocks of the address space that contain only memory which is tolerant to prefetching and speculative accesses, and that the following 4KB blocks of the address space also contain only memory which is tolerant to prefetching and speculative accesses.
- Enabling or disabling the MMU effectively changes the translation tables that are in use. The synchronization requirements that apply on changing translation tables also apply to enabling or disabling the MMU. For more information, see *Changing translation table attributes* on page B3-21. See also *Requirements for instruction caches* on page B3-23.

  In addition, if the physical address of the code that enables or disables the MMU differs from its MVA, instruction prefetching can cause complications. Therefore, ARM strongly recommends that any code that enables or disables the MMU has identical virtual and physical addresses.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

MMU が非稼動の場合は、次のいずれかの条件が満たされた場合に命令をフェッチすることができます。

- 命令が、プログラムの単純なシーケンシャル実行に必要な命令として同じ 4KB のメモリブロック（4KB にアラインされた）に存在する、または、このようなブロックに続く 4KB のメモリブロックに存在する。
- 命令が、MMU が非稼動の状態で、以前にプログラムの単純なシーケンシャル実行に必要であったものと同じ 4KB のメモリブロック（4KB にアラインされた）に存在する、または、このようなブロックに続く 4KB のメモリブロックに存在する。

注

- MMU が非稼動のときに実行される命令は、プリフェッチや投機的アクセスの影響を受けにくいメモリのみが含まれる 4KB のアドレス空間に配置され、そのアドレス空間に続く 4KB のブロックにも、プリフェッチや投機的アクセスの影響を受けにくいメモリのみが含まれることを、ソフトウェアで保証する必要があります。
- MMU の稼動または非稼動によって、使用される変換テーブルを実質的に変更することができます。変換テーブルの変更に適用される同期化要件は、MMU の稼動または非稼動にも適用されます。詳細については、*変換テーブルの属性の変更* P. B3-21 を参照して下さい。*命令キャッシュの要件* P. B3-23 も参照して下さい。

  また、MMU を稼動または非稼動にするコードの物理アドレスがその MVA と異なる場合は、命令のプリフェッチによって混乱が生じる可能性があります。このため、MMU を稼動または非稼動にするコードでは、仮想アドレスと物理アドレスを一致させることを強くお勧めします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.3 Translation tables

The MMU supports memory accesses based on memory sections or pages:

Supersections Consist of 16MB blocks of memory. Support for Supersections is optional.

Sections Consist of 1MB blocks of memory.

Large pages Consist of 64KB blocks of memory.

Small pages Consist of 4KB blocks of memory.

Support for Supersections, Sections and Large pages enables a large region of memory to be mapped using only a single entry in the TLB.

The translation tables held in memory have two levels:

First-level table

Holds *first-level descriptors* that contain the base address and

- translation properties for Sections and Supersections
- translation properties and pointers to a second level table for Large pages and Small pages

Second-level tables

Hold *second-level descriptors*, each containing the base address and translation properties for a Small pages or a Large page. Second-level tables are also referred to as *Page tables*.

The translation tables are described in the following sections:

- *Translation table entry formats*
- *Translation table base registers* on page B3-11
- *Translation table walks* on page B3-13
- *Changing translation table attributes* on page B3-21
- *The access flag* on page B3-21.

B3.3.1 Translation table entry formats

The formats of the first-level and second-level translation table descriptor entries in the translation tables are described in:

- *First-level descriptors* on page B3-8
- *Second-level descriptors* on page B3-10.

For more information about second-level translation tables see *Additional requirements for translation tables* on page B3-11.

Note

In previous versions of the *ARM Architecture Reference Manual* and in some other documentation, the AP[2] bit in the translation table entries is described as the APX bit.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.3 変換テーブル

MMU は、メモリセクションまたはページに基づくメモリアクセスをサポートします。

スーパーセクション

16MB のメモリブロックで構成されます。スーパーセクションのサポートはオプションです。

セクション 1MB のメモリブロックで構成されます。

ラージページ 64KB のメモリブロックで構成されます。

スモールページ 4KB のメモリブロックで構成されます。

スーパーセクション、セクション、およびラージページのサポートにより、TLB の単一エントリのみを使用して大きなメモリ領域をマッピングできます。

メモリに保持される変換テーブルには、次の 2 つのレベルがあります。

第 1 レベルテーブル

ベースアドレスと次のものを含む*第 1 レベル記述子*が保持されます。

- セクションとスーパーセクションに関する変換プロパティ
- ラージページとスモールページに関する変換プロパティと第 2 レベルテーブルへのポインタ

第 2 レベルテーブル

スモールページまたはラージページに関するベースアドレスと変換プロパティを含む、*第 2 レベル記述子*が保持されます。第 2 レベルテーブルは、ページテーブルとも呼ばれます。

変換テーブルについては、次のセクションで説明します。

- *変換テーブルエントリのフォーマット*
- *変換テーブル ベースレジスタ* P. B3-11
- *変換テーブルウォーク* P. B3-13
- *変換テーブルの属性の変更* P. B3-21
- *アクセスフラグ* P. B3-21

B3.3.1 変換テーブルエントリのフォーマット

変換テーブルの第 1 レベルと第 2 レベルの変換テーブル記述子エントリのフォーマットについては、次のセクションで説明します。

- *第 1 レベル記述子* P. B3-8
- *第 2 レベル記述子* P. B3-10

第 2 レベル変換テーブルの詳細については、*変換テーブルに関するその他の要件* P. B3-11 を参照して下さい。

注

旧版の『*ARM アーキテクチャ リファレンスマニュアル*』と他の一部の説明書では、変換テーブルエントリの AP[2] ビットが APX ビットと表記されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

First-level descriptors

Each entry in the first-level table is a descriptor of how the associated 1MB MVA range is mapped. Table B3-1 shows the possible first-level descriptor formats, where the value of bits [1:0] of the descriptor identifies the descriptor type:

0b00 Invalid or fault entry. The associated MVA is unmapped, and attempting to access it generates a Translation fault, see *VMSA memory aborts* on page B3-40. Software can use bits [31:2] of an invalid descriptor for its own purposes, because these bits are ignored by the hardware.

0b01 Page table descriptor. The descriptor gives the physical address of a second-level translation table, that specifies how the associated 1MByte MVA range is mapped. A second level translation table requires 1KByte of memory and can map Large pages and Small pages, see *Second-level descriptors* on page B3-10.

0b10 Section or Supersection descriptor for the associated MVA. Bit [18] determines whether the descriptor is of a Section or a Supersection. For details of how the descriptor is interpreted see *The full translation flow for Sections, Supersections, Small pages and Large pages* on page B3-15.

0b11 Reserved. In VMSAv7, descriptors with bits [1:0] == 0b11 generate Translation faults, and must not be used.

Table B3-1 VMSAv7 first-level descriptor formats

	31 … 24	23 … 20	19	18	17	16	15	14 … 12	11	10	9	8 … 5	4	3	2	1	0
Fault	IGNORE															0	0
Page table	Page table base address, bits [31:10]										IMP	Domain	SBZ	NS	SBZ	0	1
Section	Section base address, PA[31:20]		NS	0	nG	S	AP[2]	TEX[2:0]	AP[1:0]		IMP	Domain	XN	C	B	1	0
Supersection	Supersection base address PA[31:24]	Extended base address PA[35:32]	NS	1	nG	S	AP[2]	TEX[2:0]	AP[1:0]		IMP	Extended base address PA[39:36]	XN	C	B	1	0
Reserved	Reserved															1	1

The address information in the first-level descriptors is:

Page table Bits [31:10] of the descriptor are bits [31:10] of the physical address of a Page table.

Section Bits [31:20] of the descriptor are bits [31:20] of the physical address of the Section.

Supersection Bits [31:24] of the descriptor are bits [31:24] of the physical address of the Supersection. Optionally, bits [8:5,23:20] of the descriptor are bits [39:32] of the extended Supersection address.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

第 1 レベル記述子

第 1 レベルテーブルの各エントリは、関連する 1MB の MVA 範囲のマッピング方法を示す記述子です。記述子のビット [1:0] の値によって記述子タイプが識別される、第 1 レベル記述子のフォーマットを表 B3- 1 に示します。

0b00 無効なエントリまたは不正なエントリ。関連する MVA がマッピングされていないため、その MVA にアクセスしようとすると変換フォールトが発生します。*VMSA メモリアボート* P. B3- 40 を参照して下さい。無効な記述子のビット [31:2] は、ハードウェアで無視されるため、ソフトウェアで自由に使用することができます。

0b01 ページテーブル記述子。この記述子によって、関連する 1MB の MVA 範囲のマッピング方法を示す、第 2 レベル変換テーブルの物理アドレスが指定されます。第 2 レベル変換テーブルは 1KB のメモリを必要とし、ラージページとスモールページをマッピングすることができます。*第 2 レベル記述子* P. B3- 10 を参照して下さい。

0b10 関連する MVA のセクションまたはスーパーセクション記述子。ビット [18] によって、セクションとスーパーセクションのどちらの記述子かが決定されます。記述子の解釈方法については、*セクション*、*スーパーセクション*、*スモールページ*、*ラージページの完全な変換フロー* P. B3- 15 を参照して下さい。

0b11 予約。VMSAv7 では、ビット [1:0] == 0b11 の記述子によって変換フォールトが生成されるため、使用できません。

表 B3- 1 VMSAv7 の第 1 レベル記述子のフォーマット

	31 … 24	23 … 20	19	18	17	16	15	14 … 12	11	10	9	8 … 5	4	3	2	1	0
フォールト	無視															0	0
ページテーブル	ページテーブルのベースアドレス、ビット [31:10]										IMP	ドメイン	SBZ	NS	SBZ	0	1
セクション	セクションのベースアドレス、PA[31:20]		NS	0	nG	S	AP[2]	TEX[2:0]	AP[1:0]		IMP	ドメイン	XN	C	B	1	0
スーパーセクション	スーパーセクションのベースアドレス、PA[31:24]	拡張ベースアドレス、PA[35:32]	NS	1	nG	S	AP[2]	TEX[2:0]	AP[1:0]		IMP	拡張ベースアドレス、PA[39:36]	XN	C	B	1	0
予約	予約															1	1

第 1 レベル記述子のアドレス情報は次のとおりです。

ページテーブル 記述子のビット [31:10] が、ページテーブルの物理アドレスのビット [31:10] です。

セクション 記述子のビット [31:20] が、セクションの物理アドレスのビット [31:20] です。

スーパーセクション
記述子のビット [31:24] が、スーパーセクションの物理アドレスのビット [31:24] です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The other fields in the descriptors are:

TEX[2:0], C, B

Memory region attribute bits, see *Memory region attributes* on page B3-32.

These bits are not present in a Page table entry.

XN bit The execute-never bit. Determines whether the region is executable, see *The Execute Never (XN) attribute and instruction prefetching* on page B3-30.

This bit is not present in a Page table entry.

NS bit Non-secure bit. When the Security Extensions are implemented this bit specifies whether the translated PA targets Secure or Non-secure memory, see *Secure and Non-secure address spaces* on page B3-26.

Domain Domain field, see *Domains* on page B3-31.

This field is not present in a Supersection entry. Memory described by Supersections is in domain 0.

IMP bit The meaning of this bit is IMPLEMENTATION DEFINED.

AP[2], AP[1:0]

Access Permissions bits, see *Memory access control* on page B3-28.

AP[0] can be configured as the *access flag*, see *The access flag* on page B3-21.

These bits are not present in a Page table entry.

S bit The Shareable bit. Determines whether the translation is for Shareable memory, see *Memory region attributes* on page B3-32.

This bit is not present in a Page table entry.

nG bit The not global bit. Determines how the translation is marked in the TLB, see *Global and non-global regions in the virtual memory map* on page B3-54.

This bit is not present in a Page table entry.

Bit [18], when bits [1:0] == 0b10

0 Descriptor is for a Section

1 Descriptor is for a Supersection.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

オプションで、記述子のビット [8:5、23:20] が拡張スーパーセクションアドレスのビット [39:32] です。

記述子の他のフィールドは次のとおりです。

TEX[2:0]、C、B

メモリ領域属性ビット。*メモリ領域の属性* P. B3-32 を参照して下さい。

これらのビットは、ページテーブル エントリには存在しません。

XN ビット 実行不可ビット。領域が実行可能かどうかを決定します。*実行不可 (XN) 属性と命令のプリフェッチ* P. B3-30 を参照して下さい。

このビットは、ページテーブル エントリには存在しません。

NS ビット 非セキュアビット。セキュリティ拡張機能が実装されている場合、このビットは、変換された PA がセキュアメモリと非セキュアメモリのどちらかをターゲットにしているかを示します。*セキュアアドレス空間と非セキュアアドレス空間* P. B3-26 を参照して下さい。

ドメイン ドメインフィールド。*ドメイン* P. B3-31 を参照して下さい。

このフィールドは、スーパーセクションエントリには存在しません。スーパーセクションで示されるメモリは、ドメイン 0 に存在します。

IMP ビット このビットの意味は実装定義です。

AP[2]、AP[1:0]

アクセス許可ビット。*メモリアクセス制御* P. B3-28 を参照して下さい。

AP[0] は、*アクセスフラグ*として構成することができます。*アクセスフラグ* P. B3-21 を参照して下さい。

これらのビットは、ページテーブル エントリには存在しません。

S ビット 共有可能ビット。変換が共有可能メモリを対象にしているかどうかを決定します。*メモリ領域の属性* P. B3-32 を参照して下さい。

このビットは、ページテーブル エントリには存在しません。

nG ビット 非グローバルビット。変換が TLB でどのようにマークされるかを決定します。*仮想メモリマップのグローバル領域と非グローバル領域* P. B3-54 を参照して下さい。

このビットは、ページテーブル エントリには存在しません。

ビット [18]（ビット [1:0] == 0b10 の場合）

0 セクション用の記述子

1 スーパーセクション用の記述子

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Second-level descriptors

Table B3-2 shows the possible formats of a second-level descriptor, where bits [1:0] of the descriptor identify the descriptor type:

0b00 Invalid or fault entry. The associated MVA is unmapped, and attempting to access it generates a Translation fault. Software can use bits [31:2] of an invalid descriptor for its own purposes, because these bits are ignored by the hardware.

0b01 Large page descriptor. Bits [31:16] of the descriptor are the base address of the Large page.

0b1X Small page descriptor. Bits [31:12] of the descriptor are the base address of the Small page.

In this descriptor format, bit [0] of the descriptor is the XN bit.

Table B3-2 VMSAv7 second-level descriptor formats

	31 … 16	15	14 … 12	11	10	9	8	7	6	5	4	3	2	1	0
Fault	IGNORE													0	0
Large page	Large page base address, PA[31:16]		XN	TEX [2:0]		nG	S	AP [2]	SBZ		AP [1:0]		C	B	0	1
Small page	Small page base address, PA[31:12]					nG	S	AP [2]	TEX [2:0]		AP [1:0]		C	B	1	XN

The address information in the second-level descriptors is:

Large page Bits [31:16] of the descriptor are bits [31:16] of the physical address of the Large page.

Small page Bits [31:12] of the descriptor are bits [31:12] of the physical address of the Small page.

The other fields in the descriptors are:

XN bit The execute-never bit. Determines whether the region is executable, see *The Execute Never (XN) attribute and instruction prefetching* on page B3-30.

TEX[2:0], C, B

Memory region attribute bits, see *Memory region attributes* on page B3-32.

AP[2], AP[1:0]

Access Permissions bits, see *Memory access control* on page B3-28.

AP[0] can be configured as the *access flag*, see *The access flag* on page B3-21.

S bit The Shareable bit. Determines whether the translation is for Shareable memory, see *Memory region attributes* on page B3-32.

nG bit The not global bit. Used in the TLB matching process, see *Global and non-global regions in the virtual memory map* on page B3-54.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 2 レベル記述子

第 2 レベル記述子に可能なフォーマットを、表 B3-2 に示します。このフォーマットでは、記述子のビット [1:0] によって記述子タイプが識別されます。

0b00 無効または不正なエントリ。関連する MVA がマッピングされていないため、その MVA にアクセスを試みると変換フォールトが発生します。無効な記述子のビット [31:2] は、ハードウェアで無視されるため、ソフトウェアで自由に使用することができます。

0b01 ラージページ記述子。記述子のビット [31:16] は、ラージページのベースアドレスです。

0b1X スモールページ記述子。記述子のビット [31:12] は、スモールページのベースアドレスです。

この記述子フォーマットでは、ビット [0] が XN ビットです。

表 B3-2 VMSAv7 の第 2 レベル記述子のフォーマット

	31 … 16	15	14 … 12	11	10	9	8	7	6	5	4	3	2	1	0
フォールト	無視													0	0
ラージページ	ラージページのベースアドレス、PA[31:16]		XN	TEX [2:0]		nG	S	AP [2]	SBZ		AP [1:0]		C	B	0	1
スモールページ	スモールページのベースアドレス、PA[31:12]					nG	S	AP [2]	TEX[2:0]		AP [1:0]		C	B	1	XN

第 2 レベル記述子のアドレス情報は次のとおりです。

ラージページ 記述子のビット [31:16] が、ラージページの物理アドレスのビット [31:16] です。

スモールページ 記述子のビット [31:12] が、スモールページの物理アドレスのビット [31:12] です。

記述子の他のフィールドは次のとおりです。

XN ビット 実行不可ビット。領域が実行可能かどうかを決定します。*実行不可 (XN) 属性と命令のプリフェッチ* P. B3-30 を参照して下さい。

TEX[2:0]、C、B

メモリ領域属性ビット。*メモリ領域の属性* P. B3-32 を参照して下さい。

AP[2]、AP[1:0]

アクセス許可ビット。*メモリアクセス制御* P. B3-28 を参照して下さい。

AP[0] は、*アクセスフラグ*として構成することができます。*アクセスフラグ* P. B3-21 を参照して下さい。

S ビット 共有可能ビット。変換が共有可能メモリを対象にしているかどうかを決定します。*メモリ領域の属性* P. B3-32 を参照して下さい。

nG ビット 非グローバルビット。TLB 一致プロセスで使用されます。*仮想メモリマップのグローバル領域と非グローバル領域* P. B3-54 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Additional requirements for translation tables

Additional requirements for the entries in a translation table apply:

- to first-level translation tables when Supersection descriptors are used
- to second-level translation tables when Large page descriptors are used.

These requirements exist because the top four bits of the Supersection or Large page index region of the MVA overlap with the bottom four bits of the table index. *Translation table walks* on page B3-13 gives more information, and these two cases are shown in:

- Figure B3-5 on page B3-18 for the first-level translation table Supersection entry
- Figure B3-7 on page B3-20 for the second-level translation table Large page table entry.

Considering the case of using Large page table descriptors in a second-level translation table, this overlap means that for any specific Large page, the bottom four bits of the second-level translation table entry might take any value from 0b0000 to 0b1111. Therefore, each of these sixteen index values must point to a separate copy of the same descriptor. This means that, in a second-level translation table, each Large page descriptor must:

- occur first on a sixteen-word boundary
- be repeated in 16 consecutive memory locations.

For similar reasons, in a first-level translation table, each Supersection descriptor must also:

- occur first on a sixteen-word boundary
- be repeated in 16 consecutive memory locations.

Second-level translation tables are 1KB in size, and must be aligned on a 1KB boundary. Each 32-bit entry in a table provides translation information for 4KB of memory. VMSAv7 supports two page sizes:

- Large pages are 64KByte in size
- Small pages are 4KByte in size.

The required replication of Large page descriptors preserves this 4KB per entry relationship:

(4KBytes per entry) x (16 replicated entries) = 64KBytes = Large page size

B3.3.2 Translation table base registers

Three translation table registers describe the translation tables that are held in memory. For descriptions of the registers, see:

- *c2, Translation Table Base Register 0 (TTBR0)* on page B3-113
- *c2, Translation Table Base Register 1 (TTBR1)* on page B3-116
- *c2, Translation Table Base Control Register (TTBCR)* on page B3-117.

On a translation table walk, the most significant bits of the MVA and the value of TTBCR.N determine whether TTBR0 or TTBR1 is used as the translation table base register. The value of TTBCR.N indicates a number of most significant bits of the MVA and:

- if either TTBCR.N is zero or the indicated bits of the MVA are zero, TTBR0 is used
- otherwise TTBR1 is used.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

変換テーブルに関するその他の要件

次の変換テーブルのエントリには、追加の要件が適用されます。

- スーパーセクション記述子が使用されている場合の第 1 レベル変換テーブル
- ラージページ記述子が使用されている場合の第 2 レベル変換テーブル

MVA のスーパーセクションまたはラージページのインデクス領域の上位 4 ビットと、テーブルインデクスの下位 4 ビットがオーバラップすることから、これらの要件が適用されます。*変換テーブルウォーク* P. B3-13 に詳細を示します。また、この 2 つのケースについては、次のセクションで説明します。

- P. B3-18 の図 B3-5（第 1 レベル変換テーブルのスーパーセクションエントリの場合）
- P. B3-20 の図 B3-7 （第 2 レベル変換テーブルのラージページ テーブルエントリの場合）

第 2 レベル変換テーブルでラージ ページテーブル記述子を使用する場合に照らし合わせて、このオーバラップは、ラージページについて、第 2 レベル変換テーブルエントリの下位 4 ビットが 0b0000 ～ 0b1111 の値を取る可能性があることを意味します。したがって、これら 16 個のインデクス値のそれぞれは、同じ記述子の別々のコピーを指している必要があります。これは、第 2 レベル変換テーブルで、各ラージページ記述子が次の条件を満たす必要があることを意味します。

- 最初の出現が 16 ワード境界である。
- 16 個の連続メモリ位置ごとに繰り返される。

同様に、第 1 レベル変換テーブルで、各スーパーセクション記述子は次の条件を満たす必要があります。

- 最初の出現が 16 ワード境界である。
- 16 個の連続メモリ位置ごとに繰り返される。

第 2 レベル変換テーブルは、サイズが 1KB で、1KB 境界にアラインする必要があります。テーブルの各 32 ビットエントリによって、4KB のメモリに関する変換情報が提供されます。
VMSAv7 では、次の 2 つのページサイズがサポートされます。

- サイズが 64KB のラージページ
- サイズが 4KB のスモールページ

ラージページ記述子の必要な複製によって、このエントリごとに 4KB の関係が保存されます。

(エントリ単位の 4KB) × (16 個の複製されたエントリ) = 64KB = ラージページ サイズ

B3.3.2 変換テーブル ベースレジスタ

3 つの変換テーブルレジスタによって、メモリに保持された変換テーブルが記述されます。これらのレジスタの説明については、次のセクションを参照して下さい。

- *c2、変換テーブルベースレジスタ 0 (TTBR0)* P. B3-113
- *c2、変換テーブルベースレジスタ 1(TTBR1)* P. B3-116
- *c2、変換テーブルベース制御レジスタ (TTBCR)* P. B3-117

変換テーブルウォークでは、MVA の最上位ビットと TTBCR.N の値によって、TTBR0 と TTBR1 のどちらが変換テーブル ベースレジスタとして使用されるかが決定されます。TTBCR.N の値は、MVA の最上位ビットの数と、次のことを示します。

- TTBCR.N が 0 または MVA の指定されたビットが 0 の場合は、TTBR0 が使用されます。
- それ以外の場合は、TTBR1 が使用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For more information, see *Determining which TTBR to use, and the TTBR0 translation table size* on page B3-118.

The normal use of the two TTBRs is:

TTBR0 Typically used for process-specific addresses. This table ranges in size from 128bytes to 16Kbyte, depending on the value of TTBCR.N.

Each process maintains a separate first-level translation table. On a context switch:

- TTBR0 is updated to point to the first-level translation table for the new context
- TTBCR is updated if this change changes the size of the translation table
- the CONTEXTIDR is updated.

When the TTBCR is programmed to zero, all translations use TTBR0 in a manner compatible with earlier versions of the architecture, that is, with versions before ARMv6.

TTBR1 Typically used for operating system and I/O addresses, that do not change on a context switch. The size of this table is always 16KByte.

In the selected TTBR. the following bits define the memory region attributes for the translation table walk:

- the RGN, S and C bits, in the ARMv7-A base architecture
- the RGN, S, and IRGN[1:0] bits, when the Multiprocessing Extensions are implemented.

When the Security Extensions are implemented, two bits in the TTBCR for the current security state control whether a translation table walk is performed on a TLB miss:

- PD0, bit [4], controls whether translation table walks based on TTBR0 are performed
- PD1, bit [5], controls whether translation table walks based on TTBR1 are performed.

For more information about the TTBCR see *c2, Translation Table Base Control Register (TTBCR)* on page B3-117.

The effect of these bits is:

PDx == 0 When a TLB miss occurs based on TTBRx, a translation table walk is performed. The privilege of the memory access, Secure or Non-secure, corresponds to the current security state.

PDx == 1 If a TLB miss occurs based on TTBRx, a Section Translation fault is returned. No translation table walk is performed.

Note

When the Security Extensions are implemented, setting PD0 ==1 or PD1==1 can result in recursive entry into the abort handler. This effectively deadlocks the system if the mapping for the abort vectors is not guaranteed to be present in the TLB. TLB lockdown might be used to guarantee that the mapping for the abort vectors is present in the TLB.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

詳細については、*使用される TTBR と TTBR0 変換テーブルサイズの判定* P. B3-118 を参照して下さい。

2 つの TTBR の通常の用途は次のとおりです。

TTBR0 プロセス固有のアドレスとして使用されます。このテーブルのサイズは、TTBCR.N の値によって、128 バイト～ 16KB の範囲です。

プロセスごとに別々の第 1 レベル変換テーブルが保持されます。コンテキストスイッチ時には、次の処理が実行されます。

- TTBR0 が新しいコンテキストの第 1 レベル変換テーブルを指すように更新されます。
- この変更によって変換テーブルのサイズが変化した場合、TTBCR が更新されます。
- CONTEXTIDR が更新されます。

TTBCR が 0 にプログラムされている場合は、すべての変換で、以前のバージョンのアーキテクチャ、つまり、ARMv6 以前のバージョンと互換性のある方法で TTBR0 が使用されます。

TTBR1 コンテキストスイッチで変化しないオペレーティングシステムや I/O のアドレスとして使用されます。このテーブルのサイズは 16KB 固定です。

選択された TTBR では、次のビットによって変換テーブルウォーク用のメモリ領域属性が定義されます。

- RGN、S、C ビット（ARMv7-A 基本アーキテクチャの場合）
- RGN、S、IRGN[1:0] ビット（マルチプロセッシング拡張機能が実装されている場合）

セキュリティ拡張機能が実装されている場合は、現在のセキュリティ状態に対応する TTBCR の次の 2 つのビットによって、TLB ミス時に変換テーブルウォークが実行されるかどうかが制御されます。

- ビット [4] の PD0 によって、TTBR0 に基づく変換テーブルウォークが実行されるかどうかが制御されます。
- ビット [5] の PD1 によって、TTBR1 に基づく変換テーブルウォークが実行されるかどうかが制御されます。

TTBCR の詳細については、*c2、変換テーブルベース制御レジスタ (TTBCR)* P. B3-117 を参照して下さい。

これらのビットの作用は次のとおりです。

PDx == 0 TTBRx をベースとして TLB ミスが発生したときに、変換テーブルウォークが実行されます。セキュアまたは非セキュアのメモリアクセスの特権は、現在のセキュリティ状態に対応します。

PDx == 1 TTBRx をベースとして TLB ミスが発生したときに、セクション変換フォールトが返されます。変換テーブルウォークは実行されません。

注

セキュリティ拡張機能が実装されている場合は、PD0 = 1 または PD1 = 1 の設定によって、アボートハンドラが再帰的に開始される可能性があります。アボートベクタのマッピングが TLB に存在することが保証されない場合は、これによってシステムが実質的にデッドロックされます。TLB ロックダウンを使用して、アボートベクタのマッピングが TLB に存在することを保証できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.3.3 Translation table walks

A translation table walk occurs as the result of a TLB miss, and starts with a read of the appropriate first-level translation table:

- a section-mapped access only requires a read of the first-level translation table
- a page-mapped access also requires a read of the second-level translation table.

The value of the SCTLR.EE bit determines the endianness of the translation table look ups. The physical address of the base of the first-level translation table is determined from the appropriate Translation Table Base Register (TTBR), see *Translation table base registers* on page B3-11.

In the base ARMv7 architecture, and in versions of the architecture before ARMv7, it is IMPLEMENTATION DEFINED whether a hardware translation table walk can cause a read from the L1 unified or data cache. If an implementation does not support translation table accesses from L1 then software must ensure coherency between translation table walks and data updates.

Typically this involves one of:

- storing translation tables in Inner Write-Through Cacheable Normal memory
- storing translation tables in Inner Write-Back Cacheable Normal memory and ensure the appropriate cache entries are cleaned after modification
- storing translation tables in Non-cacheable memory.

For more information, see *TLB maintenance operations and the memory order model* on page B3-59.

In the Multiprocessing Extensions, translation table walks are required to access data or unified caches, or data and unified caches, of other agents participating in the coherency protocol, according to the shareability attributes described in the translation table base register. The shareability attributes described in the translation table base register must be consistent with the shareability attributes for the translation tables themselves.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.3.3 変換テーブルウォーク

TLB ミスの結果として変換テーブルウォークが発生し、該当する第 1 レベル変換テーブルの読み出しが開始されます。

- セクションマッピングされたアクセスでは、第 1 レベル変換テーブルの読み出しのみが要求されます。
- ページマッピングされたアクセスでは、第 2 レベル変換テーブルの読み出しも要求されます。

SCTLR.EE ビットの値によって、変換テーブル ルックアップのエンディアン形式が決定されます。また、第 1 レベル変換テーブルの物理ベースアドレスは、該当する変換テーブル ベースレジスタ (TTBR) から決定されます。*変換テーブル ベースレジスタ* P. B3-11 を参照して下さい。

基本 ARMv7 アーキテクチャおよび ARMv7 以前のバージョンのアーキテクチャでは、ハードウェア変換テーブルウォークによって、L1 統一またはデータキャッシュからの読み出しが発生するかどうかは実装定義です。実装で L1 からの変換テーブルアクセスがサポートされていない場合は、ソフトウェアで変換テーブルウォークとデータ更新との間のコヒーレンシを保証する必要があります。

一般に、これには次のいずれかが含まれます。

- 内部ライトスルー キャッシュ可能ノーマルメモリに変換テーブルをストアする。
- 内部ライトバック キャッシュ可能ノーマルメモリに変換テーブルをストアし、変更後に該当するキャッシュエントリがクリーニングされることを保証する。
- キャッシュ不可メモリに変換テーブルをストアする。

詳細については、*TLB 保守操作とメモリオーダ モデル* P. B3- 59 を参照して下さい。

マルチプロセッシング拡張機能では、変換テーブル ベースレジスタに記述される共有可能属性に従って、コヒーレンシプロトコルに参加している他のエージェントのデータキャッシュと統一キャッシュのどちらか、またはデータキャッシュと統一キャッシュの両方へのアクセスが変換テーブルウォークで要求されます。変換テーブル ベースレジスタで記述される共有可能属性は、変換テーブル自体の共有可能属性と一致する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Reading a first-level translation table

To perform a fetch based on TTBR1, Bits TTBR1[31:14] are concatenated with bits [31:20] of the MVA and two zero bits to produce a 32-bit physical address, as shown in Figure B3-1.

Figure B3-1 Accessing the translation table first-level descriptors based on TTBR1

When performing a fetch based on TTBR0:

- the address bits taken from TTBR0 vary between bits [31:14] and bits [31:7]
- the address bits taken from the MVA vary between bits [31:20] and bits [24:20].

The width of the TTBR0 and MVA fields depend on the value of TTBCR.N, as shown in Figure B3-2.

Figure B3-2 Accessing the translation table first-level descriptors based on TTBR0

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

第 1 レベル変換テーブルの読み出し

図 B3-1 に示すように、TTBR1 をベースとするフェッチを実行するため、ビット TTBR1[31:14]、MVA のビット [31:20]、および 2 つの 0 ビットを組み合わせて、32 ビットの物理アドレスが生成されます。

図 B3-1 TTBR1 をベースとする変換テーブルの第 1 レベル記述子へのアクセス

TTBR0 をベースとするフェッチを実行する場合、次の規則が適用されます。

- TTBR0 から取得されるアドレスビットは、ビット [31:14] とビット [31:7] の間で変動します。
- MVA から取得されるアドレスビットは、ビット [31:20] とビット [24:20] の間で変動します。

図 B3-2 に示すように、TTBR0 フィールドと MVA フィールドの幅は、TTBCR.N の値によって異なります。

図 B3-2 TTBR0 をベースとする変換テーブルの第 1 レベル記述子へのアクセス

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Regardless of which register is used as the base for the fetch, the resulting physical address selects a four-byte translation table entry that is one of:

- A first-level descriptor for a Section or Supersection.
- A *Page table* pointer to a second-level translation table. In this case a second fetch is performed to retrieve a second-level descriptor, see *Reading a second-level translation table*.
- A faulting entry.

Note

Comparing Figure B3-1 on page B3-14 with Figure B3-2 on page B3-14, you can see that when using TTBR0 with N == 0 the construction of the PA becomes identical to that for TTBR1. Other diagrams in this section show the PA formation from TTBR0, but also represent PA formation from TTBR1, for which case N = 0.

Reading a second-level translation table

Figure B3-3 shows how the address of a second-level descriptor is obtained by combining:

- the result of the first-level fetch
- the second-level table index value held in bits [19:12] of the MVA.

See Table B3-1 on page B3-8 for the format of the Access control fields of the first-level descriptor.

Figure B3-3 Accessing second-level Page table descriptors

The full translation flow for Sections, Supersections, Small pages and Large pages

This section summarizes how each of the memory section and page options is described in the translation tables, and has a subsection summarizing the full translation flow for each of the options.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

フェッチのベースとしてどのレジスタが使用されるかにかかわらず、生成された物理アドレスによって、次のいずれかの 4 バイトの変換テーブルエントリが選択されます。

- セクションまたはスーパーセクションの第 1 レベル記述子
- 第 2 レベル変換テーブルへのページテーブルポインタ。この場合、第 2 レベル記述子を読み出すために 2 回目のフェッチが実行されます。*第 2 レベル変換テーブルの読み出し*を参照して下さい。
- フォールトエントリ

注

P. B3-14 の図 B3-1 と P. B3-14 の図 B3-2 を比較すれば、N = 0 の TTBR0 を使用した場合は、PA の構造が TTBR1 の構造と同じになることがわかります。このセクションの他の図は、TTBR0 からの PA 形成を示していますが、N = 0 の場合は、TTBR1 からの PA 形成も示しています。

第 2 レベル変換テーブルの読み出し

図 B3-3 に示すように、第 2 レベル記述子のアドレスは、次のものを組み合わせて取得されます。

- 第 1 レベルフェッチの結果
- MVA のビット [19:12] に保持された第 2 レベルテーブルのインデクス値

第 1 レベル記述子のアクセス制御フィールドのフォーマットについては、P. B3-8 の表 B3-1 を参照して下さい。

図 B3-3 第 2 レベル ページテーブル記述子へのアクセス

セクション、スーパーセクション、スモールページ、ラージページの完全な変換フロー

このセクションでは、メモリセクションおよびページの各オプションが変換テーブルでどのように記述されるかを示し、サブセクションで各オプションの完全な変換フローについて説明します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The four options are:

Section A 1MB memory region, described by a first-level translation table descriptor with bits [18,1:0] == 0b010.

See *Translation flow for a Section* on page B3-17.

Supersection A 16MB memory region, described by a first-level translation table entry with bits [18,1:0] == 0b110.

See *Translation flow for a Supersection* on page B3-18.

Small page A 4KB memory region, described by:

- a first-level translation table entry with bits [1:0] == 0b01, giving a second-level Page table address.
- a second-level descriptor with bit [1] == 1.

See *Translation flow for a Small page* on page B3-19.

Large page A 64KB memory region, described by:

- a first-level translation table entry with bits [1:0] == 0b01, giving a second-level Page table address.
- a second-level descriptor with bits [1:0] == 0b01.

See *Translation flow for a Large page* on page B3-20.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

オプションは次の 4 つです。

セクション ビット [18、1:0] == 0b010 の第 1 レベル変換テーブル記述子により記述される 1MB のメモリ領域。

詳細については、*セクションの変換フロー* P. B3- 17 を参照して下さい。

スーパーセクション

ビット [18、1:0] == 0b110 の第 1 レベル変換テーブルエントリにより記述される 16MB のメモリ領域。

詳細については、*スーパーセクションの変換フロー* P. B3- 18 を参照して下さい。

スモールページ

次のものによって示される 4KB のメモリ領域。

- 第 2 レベル ページテーブルアドレスを提供する、ビット [1:0] == 0b01 の第 1 レベル変換テーブルエントリ。
- ビット [1] == 1 の第 2 レベル記述子

詳細については、*スモールページの変換フロー* P. B3- 19 を参照して下さい。

ラージページ

次のものによって示される 64KB のメモリ領域。

- 第 2 レベル ページテーブルアドレスを提供する、ビット [1:0] == 0b01 の第 1 レベル変換テーブルエントリ。
- ビット [1:0] == 0b01 の第 2 レベル記述子

詳細については、*ラージページの変換フロー* P. B3- 20 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Translation flow for a Section

Figure B3-4 shows the virtual to physical addresses translation for a Section. For details of the access control fields in the first-level descriptor see the *Section* entry in Table B3-1 on page B3-8.

For a translation based on TTBR0, N is the value of TTBCR.N

For a translation based on TTBR1, N is 0

‡ This field is absent if N==0

Figure B3-4 Section address translation

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

セクションの変換フロー

セクションの仮想アドレスから物理アドレスへの変換を、図 B3-4 に示します。第 1 レベル記述子のアクセス制御フィールドの詳細については、P. B3-8 の表 B3-1 の *セクション*のエントリを参照して下さい。

TTBR0に基づく変換では、NはTTBCR.Nの値

TTBR1に基づく変換では、N==0

‡ N==0の場合は、このフィールドが存在しない

図 B3-4 セクションアドレスの変換

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Translation flow for a Supersection

Figure B3-5 shows the virtual to physical addresses translation for a Supersection. For details of the extended Supersection address and access control fields in the first-level descriptor see the *Supersection* entry in Table B3-1 on page B3-8.

For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
‡ This field is absent if N==0

Figure B3-5 Supersection address translation

Note

Figure B3-5 shows how, when the MVA addresses a Supersection, the top four bits of the *Supersection index* bits of the MVA overlap the bottom four bits of the *Table index* bits. For more information, see *Additional requirements for translation tables* on page B3-11.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

スーパーセクションの変換フロー

スーパーセクションの仮想アドレスから物理アドレスへの変換を、図 B3-5 に示します。第 1 レベル記述子の拡張スーパーセクションアドレスおよびアクセス制御フィールドの詳細については、P. B3-8 の表 B3-1 の *スーパーセクション*のエントリを参照して下さい。

TTBR0に基づく変換では、NはTTBCR.Nの値
TTBR1に基づく変換では、N==0
‡ N==0の場合は、このフィールドが存在しない

図 B3-5 スーパーセクションアドレスの変換

注

MVA によってスーパーセクションがアドレス指定されている場合に、MVA の *スーパーセクションインデクス* ビットの上位 4 ビットと、*テーブルインデクス*ビットの下位 4 ビットがどのようにオーバラップするかを、図 B3-5 に示します。詳細については、*変換テーブルに関するその他の要件* P. B3-11 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Translation flow for a Small page

Figure B3-6 shows the virtual to physical addresses translation for a Small page. For details of the access control fields in the first-level descriptor see the *Page table* entry in Table B3-1 on page B3-8. For details of the access control fields in the second-level descriptor see the *Small page* entry in Table B3-2 on page B3-10.

For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
‡ This field is absent if N==0

Figure B3-6 Small page address translation

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

スモールページの変換フロー

スモールページの仮想アドレスから物理アドレスへの変換を、図 B3-6 に示します。第 1 レベル記述子のアクセス制御フィールドの詳細については、P. B3-8 の表 B3-1 のページテーブルのエントリを参照して下さい。第 2 レベル記述子のアクセス制御フィールドの詳細については、P. B3-10 の表 B3-2 のスモールページのエントリを参照して下さい。

TTBR0に基づく変換では、NはTTBCR.Nの値
TTBR1に基づく変換では、N==0
‡ N==0の場合は、このフィールドが存在しない

図 B3-6 スモールページ アドレスの変換

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Translation flow for a Large page

Figure B3-7 shows the virtual to physical addresses translation for a Large page. For details of the access control fields in the first-level descriptor see the *Page table* entry in Table B3-1 on page B3-8. For details of the access control fields in the second-level descriptor see the *Large page* entry in Table B3-2 on page B3-10.

For a translation based on TTBR0, N is the value of TTBR0.N
For a translation based on TTBR1, N is 0
‡ This field is absent if N==0

Figure B3-7 Large page address translation

Note

Figure B3-7 shows how, when the MVA addresses a Large page, the top four bits of the *page index* bits of the MVA overlap the bottom four bits of the *First-level table index* bits. For more information, see *Additional requirements for translation tables* on page B3-11. This diagram also shows the width of the *page index* bits when addressing a Small page, to show that there is no overlap in this case.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ラージページの変換フロー

ラージページの仮想アドレスから物理アドレスへの変換を、図 B3-7 に示します。第 1 レベル記述子のアクセス制御フィールドの詳細については、P. B3-8 の表 B3-1 のページテーブルのエントリを参照して下さい。第 2 レベル記述子のアクセス制御フィールドの詳細については、P. B3-10 の表 B3-2 のラージページのエントリを参照して下さい。

TTBR0に基づく変換では、NはTTBCR.Nの値
TTBR1に基づく変換では、N==0
‡ N==0の場合は、このフィールドが存在しない

図 B3-7 ラージページ アドレスの変換

注

MVA によってラージページがアドレス指定されている場合に、MVA のページインデクスビットの上位 4 ビットと、*第 1 レベル テーブルインデクス*ビットの下位 4 ビットとがどのようにオーバラップするかを、図 B3-7 に示します。詳細については、*変換テーブルに関するその他の要件* P. B3-11 を参照して下さい。この図は、スモールページがアドレス指定されている場合のページインデクスビットの幅も示しており、このケースではオーバラップが存在しないことがわかります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.3.4 Changing translation table attributes

When changing translation table attributes, you must avoid situations where caching or pipelining effects mean that overlapping entries or aliases with different attributes might be visible to the processor at the same time. To avoid these situations, ARM recommends that you invalidate old translation table entries, and synchronize the effects of those invalidations, before you create new translation table entries that might overlap or create aliases with different attributes. This approach is sometimes called *break before make*.

For information about the procedure for synchronizing a change to the translation tables see *TLB maintenance operations and the memory order model* on page B3-59.

Translation table entries that create Translation faults are not held in the TLB, see *Translation fault* on page B3-43. Therefore TLB and branch predictor invalidation is not required for the synchronization of a change from a translation table entry that causes a Translation fault to one that does not.

B3.3.5 The access flag

From VMSAv7, the AP[0] bit in the translation table descriptors can be redefined as an access flag. This is done by setting SCTLR.AFE to 1, see *c1, System Control Register (SCTLR)* on page B3-96. When this bit is set, the access permissions information in the translation table descriptors is limited to the AP[2:1] bits, as described in *Simplified access permissions model* on page B3-29.

The access flag is used to indicate when a page or section of memory is accessed for the first time since the access flag was set to 0.

It is IMPLEMENTATION DEFINED whether the access flag is managed by software or by hardware. The two options are described in the subsections:

- *Software management of the access flag*
- *Hardware management of the access flag*.

The access flag mechanism expects that, when an Access Flag fault occurs, software sets the access flag to 1 in the translation table entry that caused the fault. This prevents the fault occurring the next time the memory is accessed. Software does not have to flush the entry from the TLB after setting the flag.

Software management of the access flag

With an implementation that requires software to manage the access flag, an Access Flag fault is generated when both:

- the SCTLR.AFE bit is set to 1
- a translation table entry with the access flag set to 0 is read into the TLB.

Hardware management of the access flag

An implementation can choose to provide hardware management of the access flag. In this case, when the SCTLR.AFE bit is set to 1 and a translation table entry with the access flag set to 0 is read into the TLB, the hardware must write 1 to the access flag bit of the translation table entry in memory.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.3.4 変換テーブルの属性の変更

変換テーブルの属性を変更するときは、キャッシュまたはパイプラインの動作により、オーバラップしているエントリ、または異なる属性を持つエイリアスがプロセッサから同時に認識可能になる状況を避ける必要があります。この状況を避けるためには、オーバラップする可能性がある新たな変換テーブルエントリ、または異なる属性を持つエイリアスを作成する前に、古い変換テーブルエントリを無効にして、その無効化の影響を同期することをお勧めします。この手法は、*ブレーク ビフォーメイク*と呼ばれることがあります。

変換テーブルの変更を同期する手順の詳細については、*TLB 保守操作とメモリオーダ モデル* P. B3-59 を参照して下さい。

変換フォールトを生成した変換テーブルエントリは、TLB 内に保持されません。*変換フォールト* P. B3-43 を参照して下さい。したがって、変換フォールトを引き起こした変換テーブルエントリから、それ以外の変換テーブルエントリへの変更の同期では、TLB と分岐予測器を無効にする必要がありません。

B3.3.5 アクセスフラグ

VMSAv7 およびそれ以降では、変換テーブル記述子の AP[0] ビットをアクセスフラグとして再定義することができます。これは、SCTLR.AFE を 1 にセットすることによって実行されます。*c1、システム制御レジスタ (SCTLR)* P. B3-96 を参照して下さい。*単純なアクセス許可モデル* P. B3-29 で説明されているように、このビットがセットされている場合は、変換テーブル記述子のアクセス許可情報が AP[2:1] ビットに制限されます。

アクセスフラグは、0 に設定された後で最初にメモリのページまたはセクションがアクセスされた時点を示すために使用されます。

アクセスフラグがソフトウェアとハードウェアのどちらによって管理されるかは実装定義です。これら 2 つのオプションについては、次のサブセクションで説明します。

- *アクセスフラグのソフトウェア管理*
- *アクセスフラグのハードウェア管理*

アクセスフラグ機構は、アクセスフラグ フォールトが発生したときに、そのフォールトを引き起こした変換テーブルエントリのアクセスフラグが、ソフトウェアにより 1 にセットされることを前提としています。これによって、次にメモリがアクセスされたときにフォールトの発生が回避されます。ソフトウェアでは、このフラグをセットした後で、TLB からエントリをフラッシュする必要はありません。

アクセスフラグのソフトウェア管理

アクセスフラグをソフトウェアで管理する必要のある実装では、次の両方が成立した場合にアクセスフラグ フォールトが生成されます。

- SCTLR.AFE ビットが 1 にセットされている。
- アクセスフラグが 0 にセットされた変換テーブルエントリが TLB に読み出される。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Any implementation of hardware management of the access flag must ensure that any software changes to the translation table are not lost. The architecture does not require software that performs translation table changes to use interlocked operations. The hardware management mechanisms for the access flag must prevent any loss of data written to translation table entries that might occur when, for example, a write by another processor occurs between the read and write phases of a translation table walk that updates the access flag.

An implementation that provides hardware management of the access flag:

- does not generate Access Flag faults when the access flag is enabled
- uses the HW access flag field, ID_MMFR2[31:28], to indicate this implementation choice, see *c0, Memory Model Feature Register 2 (ID_MMFR2)* on page B5-14.

Architecturally, an operating system that makes use of the access flag must support the software faulting option that uses the Access Flag fault. This provides compatibility between systems that include a hardware implementation of the access flag and those systems that do not implement this feature.

When an implementation provides hardware management of the access flag it must also implement the SCTLR.HA bit, that can be used to enable or disable the access flag mechanism. See *c1, System Control Register (SCTLR)* on page B3-96.

Changing the access flag enable

It is UNPREDICTABLE whether the TLB caches the effect of the SCTLR.AFE bit on translation tables. This means that, after changing the SCTLR.AFE bit software must invalidate the TLB before it relies on the effect of the new value of the SCTLR.AFE bit.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アクセスフラグのハードウェア管理

実装で、アクセスフラグのハードウェア管理を提供することができます。この場合、SCTLR.AFE ビットが 1 にセットされているときに、アクセスフラグが 0 にセットされた変換テーブルエントリが TLB に読み出された場合、メモリの変換テーブルエントリのアクセスフラグ ビットに、ハードウェアが 1 を書き込む必要があります。

アクセスフラグのハードウェア管理実装では、ソフトウェアによる変換テーブルの変更が失われないことを保証する必要があります。アーキテクチャ上は、ソフトウェアで、インターロックされた操作を使用するために変換テーブルを変更する必要はありません。アクセスフラグのハードウェア管理機構では、例えば、アクセスフラグを更新する変換テーブルウォークの読み出しフェーズと書き込みフェーズとの間で別のプロセッサによる書き込みが実行された場合に発生する可能性のある、変換テーブルエントリに書き込まれたデータの消失を避ける必要があります。

アクセスフラグのハードウェア管理を提供する実装では、次の方法を使用します。

- アクセスフラグが可能になっている場合はアクセスフラグ フォールトを生成しない。
- この実装の選択を示すために、ハードウェアのアクセスフラグ フィールド (ID_MMFR2[31:28]) を使用する。*c0、メモリモデル機能レジスタ 2 (ID_MMFR2)* P. B5-14 を参照して下さい。

アーキテクチャ上は、アクセスフラグを使用するオペレーティングシステムは、アクセスフラグ フォールトを使用するソフトウェアフォールト オプションをサポートする必要があります。これによって、アクセスフラグのハードウェア実装を含むシステムと、その機能を実装していないシステムとの互換性が維持されます。

実装でアクセスフラグのハードウェア管理を提供する場合は、アクセスフラグ機構の可能 / 不可能を切り替えるための SCTLR.HA ビットも実装する必要があります。詳細については、*c1、システム制御レジスタ (SCTLR)* P. B3-96 を参照して下さい。

アクセスフラグ イネーブルの変更

変換テーブルに対する SCTLR.AFE ビットの作用が TLB でキャッシュされるかどうかは、予測不能です。これは、SCTLR.AFE ビットを変更した後で、ソフトウェアは SCTLR.AFE ビットの新しい値の影響を前提とする前に、TLB の無効化を行う必要があることを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.4 Address mapping restrictions

ARMv6 supported a *page coloring* restriction that, when implemented, required all Virtual Address aliases of a given Physical Address to have the same value for address bits [13:12]. This page coloring restriction was required to support *Virtually Index Physically Tagged* (VIPT) caches with a cache way size larger than 4KBytes. For details of the page coloring restriction see *Virtual to physical translation mapping restrictions* on page AppxG-26.

ARMv7 does not support page coloring, and requires that all data and unified caches behave as *Physically Indexed Physically Tagged* (PIPT) caches.

Note

An ARMv7 implementation might use techniques such as hardware alias avoidance to make a VIPT cache behave as a PIPT cache, and might improve performance by avoiding accesses to frequently alternating aliases to a physical address. Such approaches give good results, but ARM recommends migration to the use of true PIPT caches for all data and unified caches.

In an ARMv7 implementation, any data or unified cache maintenance operation that operates on a virtual address must take account of the fact that the cache behaves as a PIPT cache. This means that the implementation must perform the appropriate action on the physical address that corresponds to the MVA targeted by the operation.

The ARMv7 requirements for instruction caches are described in *Requirements for instruction caches*.

B3.4.1 Requirements for instruction caches

In a base VMSAv7 implementation, the following conditions require cache maintenance of an instruction cache:

- writing new data to an instruction address
- writing new address mappings to the translation table
- changing one or more of the TTBR0, TTBR1 and TTBCR registers without changing the ASID
- enabling or disabling the MMU, by writing to the SCTLR.

Note

These conditions are consistent with the maintenance required for an ASID-tagged Virtually Indexed Virtually Tagged (VIVT) instruction cache that also includes a security status bit for each cache entry.

VMSAv7 can be implemented with an optional extension, the *IVIPT extension* (Instruction cache Virtually Indexed Physically Tagged extension). The effect of this extension is to reduce the instruction cache maintenance requirement to a single condition:

- writing new data to an instruction address.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.4 アドレスマッピングの制限

ARMv6 では、*ページのカラーリング*が実装されている場合に、特定の物理アドレスに対するすべての仮想アドレスエイリアスのアドレスビット [13:12] の値を同じにしなければならないという制限がサポートされていました。このページのカラーリング制限は、キャッシュウェイ サイズが 4KB を超える*仮想インデクス物理タグ付き*(VIPT) キャッシュをサポートするために必要でした。ページのカラーリング制限の詳細については、*仮想アドレスから物理アドレスへの変換マッピングに関する制約条件* P. 付録 G-26 を参照して下さい。

ARMv7 ではページのカラーリングがサポートされていないため、すべてのデータキャッシュと統一キャッシュが*物理インデクス物理タグ付き*(PIPT) キャッシュとして動作する必要があります。

注

ARMv7 実装では、ハードウェアエイリアス回避などの技法を使用して VIPT キャッシュを PIPT キャッシュとして動作させたり、頻繁に変更される物理アドレスへのエイリアスにアクセスすることを避け、パフォーマンスを向上させたりすることができます。このような手法は良好な結果が得られますが、すべてのデータキャッシュと統一キャッシュに対する真の PIPT キャッシュの使用への移行をお勧めします。

ARMv7 実装では、仮想アドレスに対するすべてのデータまたは統一キャッシュ保守操作で、キャッシュが PIPT キャッシュとして動作するという事実を考慮する必要があります。これは、実装で、操作対象の MVA に対応する物理アドレスに対して適切なアクションを実行する必要があることを意味します。

ARMv7 の命令キャッシュに関する要件については、*命令キャッシュの要件*で説明します。

B3.4.1 命令キャッシュの要件

基本 VMSAv7 実装では、次の場合に命令キャッシュの保守が必要です。

- 命令アドレスへの新しいデータの書き込み
- 変換テーブルへの新しいアドレスマッピングの書き込み
- ASID に影響を与えない TTBR0、TTBR1、TTBCR レジスタの変更
- SCTLR への書き込みによる MMU の許可または禁止

注

これらの条件は、キャッシュエントリごとにセキュリティ ステータスビットを含む、ASID タグ付き仮想インデクス仮想タグ付き (VIVT) 命令キャッシュに必要な保守と一致します。

VMSAv7 には、オプション拡張機能の *IVIPT 拡張機能*（命令キャッシュの仮想インデクス物理タグ付き拡張機能）を実装できます。この拡張機能によって、命令キャッシュの保守要件は、次の 1 つの条件に削減されます。

- 命令アドレスへの新しいデータの書き込み

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

This condition is consistent with the maintenance required for a Virtually Indexed Physically Tagged (VIPT) instruction cache.

Software can read the Cache Type Register to determine whether the IVIPT extension is implemented, see *c0, Cache Type Register (CTR)* on page B3-83.

Functionally, the relationship between cache type and the software management requirement depends on whether the operating system uses ASIDs to distinguish processes that use different translation tables:

- when ASIDs are used, management is similar for a VIPT and an ASID-tagged VIVT cache
- when ASIDs are not used, management is similar for a VIVT and an ASID-tagged VIVT cache.

A remapping policy that supports ASID changes means that translation tables can be swapped simply by updates to the TTBR0, TTBR1 and TTBCR registers, with an appropriate change of the ASID held in the CONTEXTIDR, see *Synchronization of changes of ASID and TTBR* on page B3-60. Such changes are transparent to an ASID-tagged VIVT instruction cache until an ASID value is reused. In contrast, a VIVT instruction cache that is not ASID-tagged must be invalidated whenever the virtual to physical address mappings change. Therefore, such a cache must be invalidated on an ASID change.

Software written to rely on a VIPT instruction cache must only be used with processors that implement the IVIPT. For maximum compatibility across processors, ARM recommends that operating systems target the ARMv7 base architecture that uses ASID-tagged VIVT instruction caches, and do not assume the presence of the IVIPT extension. Software that relies on the IVIPT extension might fail in an UNPREDICTABLE way on an ARMv7 implementation that does not include the IVIPT extension.

With an instruction cache, the distinction between a VIPT cache and a PIPT cache is much less visible to the programmer than it is for a data cache, because normally the contents of an instruction cache are not changed by writing to the cached memory. However, there are situations where a program must distinguish between the different cache tagging strategies. Example B3-1 describes such a situation.

Example B3-1 A situation where software must be aware of the Instruction cache tagging strategy

Two processes, P_1 and P_2, share some code and have separate virtual mappings to the same region of instruction memory. P_1 changes this region, for example as a result of a JIT, or some other self-modifying code operation. P_2 needs to see the modified code.

As part of its self-modifying code operation, P_1 must invalidate the changed locations from the instruction cache. For more information, see *Ordering of cache and branch predictor maintenance operations* on page B2-21. If this invalidation is performed by MVA, and the instruction cache is a VIPT cache, then P_2 might continue to see the old code.

In this situation, if the instruction cache is a VIPT cache, after the code modification the entire instruction cache must be invalidated to ensure P_2 observes the new version of the code.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

この条件は、仮想インデクス物理タグ付き (VIPT) 命令キャッシュに必要な保守と一致します。

ソフトウェアでキャッシュタイプ レジスタを読み出して、IVIPT 拡張機能が実装されているかどうかを判断できます。*c0、キャッシュタイプ レジスタ (CTR)* P. B3-83 を参照して下さい。

機能上、キャッシュタイプとソフトウェア管理要件との関係は、別々の変換テーブルを使用しているプロセスをオペレーティングシステムで区別するために ASID が使用されているかどうかによって異なります。

- ASID が使用されている場合は、VIPT キャッシュと ASID タグ付き VIVT キャッシュに対する管理はほぼ同じです。
- ASID が使用されていない場合は、VIVT キャッシュと ASID タグ付き VIVT キャッシュに対する管理はほぼ同じです。

ASID 変更をサポートする再マッピングポリシーは、CONTEXTIDR 内に保持された該当する ASID の適切な変更、および TTBR0、TTBR1、TTBCR レジスタの更新によって、変換テーブルを交換できることを意味します。*ASID と TTBR の変更の同期化* P. B3-60 を参照して下さい。このような変更は、ASID 値が再使用されるまで、ASID タグ付き VIVT 命令キャッシュから透過的です。これに対して、仮想アドレスから物理アドレスへのマッピングが変更された場合は、ASID タグ付きでない VIVT 命令キャッシュを無効化する必要があります。したがって、このようなキャッシュは ASID の変更時に無効化する必要があります。

VIPT 命令キャッシュに依存するように作成されたソフトウェアは、IVIPT を実装したプロセッサでのみ使用する必要があります。プロセッサ間で最大限の互換性を維持するために、オペレーティングシステムは、ASID タグ付き VIVT 命令キャッシュを使用する ARMv7 基本アーキテクチャを対象とし、IVIPT 拡張機能の存在を前提としないことをお勧めします。IVIPT 拡張機能に依存するソフトウェアは、IVIPT 拡張機能を含まない ARMv7 実装上では予測不能な結果を招く可能性があります。

命令キャッシュでは一般に、キャッシュされたメモリへの書き込みによって命令キャッシュの内容が変更されることはないため、VIPT キャッシュと PIPT キャッシュとの違いが、データキャッシュほどプログラマに認識されません。ただし、プログラムで複数のキャッシュタグ付け方式を区別しなければならない場合があります。このような状況については、例 B3-1 で説明します。

例 B3-1 ソフトウェアで命令キャッシュタグ付け方式を認識しなければならない状況

P_1 と P_2 の 2 つのプロセスで、一部のコードが共有され、命令メモリの同じ領域に対して別々の仮想マッピングが設定されています。P_1 で、例えば JIT やその他の自己修正コード操作の結果として、この領域が変更されます。P_2 では、変更されたコードが認識される必要があります。

自己修正コード処理の一部として、P_1 は、命令キャッシュから変更された位置を無効化する必要があります。詳細については、*キャッシュの保守操作と分岐予測器の保守操作との順序付け* P. B2-21 を参照して下さい。この無効化が MVA によって実行され、命令キャッシュが VIPT キャッシュの場合は、P_2 で古いコードが認識され続ける可能性があります。

この状況で、命令キャッシュが VIPT キャッシュの場合は、コードの変更後に命令キャッシュ全体を無効化して、P_2 で新しいバージョンのコードが観測されることを保証する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

Software can read the Cache Type Register to determine whether the instruction cache is PIPT, VIPT, or ASID-tagged VIVT, see *c0, Cache Type Register (CTR)* on page B3-83.

B3.4.2 Instruction cache maintenance operations by MVA

On a cache maintenance operation by MVA, the generation of Data Abort exceptions can depend on the tagging strategy of the instruction cache:

- With an ASID-tagged VIVT instruction cache, it is IMPLEMENTATION DEFINED whether the TLB is checked to see whether a valid translation table mapping exists for the VA used by a cache maintenance operation. Therefore, it is IMPLEMENTATION DEFINED whether cache maintenance operations by MVA can generate Data Abort exceptions.
- With a VIPT or PIPT instruction cache, the TLB must be checked and an abort is generated on a Translation fault or an Access Flag fault. No abort is generated on a Domain or Permission fault.

For maximum portability, ARM recommends that operating systems always provide an abort handler to process Data Abort exceptions on instruction cache maintenance operations by MVA, even though some ARMv7 implementations might not be capable of generating these aborts.

The effect of an instruction cache maintenance operation by MVA can depend on the tagging strategy of the instruction cache:

- For an ASID -tagged VIVT instruction cache or a VIPT instruction cache, the effect of the operation is only guaranteed to apply to the modified virtual address supplied to the instruction. It is not guaranteed to apply to any other alias of that modified virtual address.
- For a PIPT instruction cache, the effect of the operation applies to all aliases of the modified virtual address supplied to the instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

ソフトウェアでキャッシュタイプ レジスタを読み出して、命令キャッシュが PIPT、VIPT、ASID タグ付き VIVT のどれであるかを判断できます。*c0、キャッシュタイプ レジスタ (CTR)* P. B3- 83 を参照して下さい。

B3.4.2 MVA による命令キャッシュ保守操作

MVA によるキャッシュ保守操作では、データアボート例外の生成が命令キャッシュのタグ付け方式によって異なる可能性があります。

- ASID タグ付き VIVT 命令キャッシュを使用している場合は、TLB をチェックして、キャッシュ保守操作で使用される VA について有効な変換テーブルマッピングが存在することを確認するかどうかは実装定義です。したがって、MVA によるキャッシュ保守操作がデータアボート例外を生成するかどうかは実装定義です。
- VIPT または PIPT 命令キャッシュを使用している場合は、TLB をチェックする必要があります。変換フォールトまたはアクセスフラグ フォールトでアボートが生成されます。ドメインフォールトまたはアクセス許可フォールトではアボートが生成されません。

最大限の移植性を維持するために、ARMv7 実装によっては MVA による命令キャッシュ保守操作でのデータアボート例外が生成されない場合がありますが、オペレーティングシステムでは常に、このようなデータアボート例外を処理するアボートハンドラを提供することをお勧めします。

MVA による命令キャッシュ保守操作の作用は、命令キャッシュのタグ付け方式によって異なる可能性があります。

- ASID タグ付き VIVT 命令キャッシュまたは VIPT 命令キャッシュの場合は、この操作の作用は、命令に渡された修飾仮想アドレスにのみ適用が保証されます。この修飾仮想アドレスの他のエイリアスに対する適用は保証されません。
- PIPT 命令キャッシュの場合は、操作の作用が、命令に渡された変更済みの仮想アドレスのすべてのエイリアスに適用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.5 Secure and Non-secure address spaces

When implemented, the Security Extensions provide two physical address spaces, a Secure physical address space and a Non-secure physical address space.

The translation table base registers, TTBR0, TTBR1 and TTBCR are banked between Secure and Non-secure versions, and the security state of a memory access selects the corresponding version of the registers. Therefore, the translation tables are separated between Secure and Non-secure versions. Translation table walks are made to the physical address space corresponding to the security state of the translation tables.

The Non-secure translation table entries can only translate a virtual address to a physical address in the Non-secure physical address space. Secure translation table entries can translate a virtual address to a physical addresses in either the Secure or the Non-secure address space. Selection of which physical address space to use is managed by the NS field in the first-level descriptors, see *First-level descriptors* on page B3-8:

- for Non-secure translation table entries, the NS field is ignored
- for Secure translation table entries, the NS field determines which physical address space is accessed:

  NS == 0 Secure physical address space is accessed.

  NS == 1 Non-secure physical address space is accessed.

Because the NS field is defined only in the first level translation tables, the granularity of the Secure and Non-secure memory spaces is 1MB. However, in these memory regions you can define physical memory regions with a granularity of 4KB. For more information, see *Translation tables* on page B3-7.

Note

A system implementation can alias parts of the Secure physical address space to the Non-secure physical address space in implementation-specific ways. As with any other aliasing of physical memory, the use of aliases in this way can require the use of cache maintenance operations to ensure that changes to memory made using one alias of the physical memory are visible to accesses to the other alias of the physical memory.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.5 セキュアアドレス空間と非セキュアアドレス空間

セキュリティ拡張機能が実装されている場合は、セキュア物理アドレス空間と非セキュア物理アドレス空間の 2 つの物理アドレス空間が提供されます。

変換テーブル ベースレジスタの TTBR0、TTBR1、TTBCR はセキュアバージョンと非セキュアバージョンとの間でバンクされ、メモリアクセスのセキュリティ状態によって、対応するバージョンのレジスタが選択されます。そのため、変換テーブルがセキュアバージョンと非セキュアバージョンとで分割されます。変換テーブルウォークは、変換テーブルのセキュリティ状態に対応する物理アドレス空間に対して実行されます。

非セキュア変換テーブルエントリは、仮想アドレスを、非セキュア物理アドレス空間の物理アドレスに変換することしかできません。セキュア変換テーブルエントリは、仮想アドレスを、セキュアまたは非セキュアのアドレス空間内の物理アドレスに変換できます。使用される物理アドレス空間の選択は、第 1 レベル記述子の NS フィールドによって管理されます。*第 1 レベル記述子* P. B3-8 を参照して下さい。

- 非セキュア変換テーブルエントリの場合、NS フィールドは無視されます。
- セキュア変換テーブルエントリの場合、NS フィールドによってアクセス対象の物理アドレス空間が決定されます。

  NS == 0 セキュア物理アドレス空間がアクセスされます。

  NS == 1 非セキュア物理アドレス空間がアクセスされます。

NS フィールドは第 1 レベル変換テーブルにのみ定義されるため、セキュアと非セキュアのメモリ空間の粒度は 1MB です。ただし、これらのメモリ領域では、4KB の粒度で物理メモリ領域を定義することができます。詳細については、*変換テーブル* P. B3-7 を参照して下さい。

注

システム実装固有の方法で、セキュア物理アドレス空間の一部を非セキュア物理アドレス空間にエイリアスすることができます。その他の物理メモリのエイリアシングと同様に、この方法でエイリアスを使用する場合は、キャッシュ保守操作により、ある物理メモリエイリアスを使用したメモリ変更が、他の物理メモリエイリアスへのアクセスから可視であることを保証する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.5.1 The effect of the Security Extensions on the cache operations

When the Security Extensions are implemented and each security state has its own physical address space, Table B3-3 shows the effect of the security state on the cache operations.

Table B3-3 Effect of the security state on the cache operations

Cache operation	Security state	Targeted entry
Instruction cache operations		
Invalidate All	Non-secure	All instruction cache lines that contain entries that can be accessed from the Non-secure security state
Invalidate All	Secure	All instruction cache lines
Invalidate by MVA	Either	Base Architecture: All Lines that match the specified MVA and the current ASID and come from the same virtual address space as the current security state
		IVIPT extension:[a] All Lines that match the specified MVA and the current ASID and come from the same physical address space as described in the translation tables
Data or unified cache operations		
Invalidate, Clean, Clean and Invalidate by set/way	Non-secure	Line specified by set/way provided that the entry comes from the Non-secure physical address space
Invalidate, Clean, Clean and Invalidate by set/way	Secure	Line specified by set/way regardless of the physical address space that the entry has come from
Invalidate, Clean, Clean and Invalidate by MVA	Either	All Lines that match the specified MVA and the current ASID and come from the same physical address space, as described in the translation tables

a. For more information about the IVIPT extension see *Requirements for instruction caches* on page B3-23.

For locked entries and entries that might be locked, the behavior of cache maintenance operations described in *The interaction of cache lockdown with cache maintenance* on page B2-18 applies. This behavior is not affected by the Security Extensions.

With an implementation that generates aborts if entries are locked or might be locked in the cache, if the use of lockdown aborts is enabled then these aborts can occur on any cache maintenance operation regardless of the Security Extensions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.5.1 キャッシュ操作に対するセキュリティ拡張機能の作用

セキュリティ拡張機能が実装されており、セキュリティ状態ごとに物理アドレス空間が割り当てられている場合の、キャッシュ操作に対するセキュリティ状態の作用を表 B3-3 に示します。

表 B3-3 キャッシュ操作に対するセキュリティ状態の作用

キャッシュ操作	セキュリティ状態	対象のエントリ
命令キャッシュ操作		
全体の無効化	非セキュア	非セキュアセキュリティ状態からアクセス可能なエントリを含む、すべての命令キャッシュライン
全体の無効化	セキュア	すべての命令キャッシュライン
MVA による無効化	いずれか	基本アーキテクチャ 指定された MVA および現在の ASID と一致し、現在のセキュリティ状態と同じ仮想アドレス空間に含まれるすべてのライン
		IVIPT 拡張機能[a] 指定された MVA および現在の ASID と一致し、変換テーブルに記述されているものと同じ物理アドレス空間に含まれるすべてのライン
データまたは統一キャッシュ操作		
セット / ウェイによる無効化、クリーニング、クリーニングと無効化	非セキュア	エントリが非セキュア物理アドレス空間に含まれる場合に、セット / ウェイによって指定されたライン
セット / ウェイによる無効化、クリーニング、クリーニングと無効化	セキュア	エントリが含まれる物理アドレス空間に関係なく、セット / ウェイによって指定されたライン
MVA による無効化、クリーニング、クリーニングと無効化	いずれか	指定された MVA および現在の ASID と一致し、変換テーブルに記述されているものと同じ物理アドレス空間に含まれるすべてのライン

a. IVIPT 拡張機能の詳細については、*命令キャッシュの要件* P. B3-23 を参照して下さい。

ロックされたエントリとロックされる可能性のあるエントリの場合は、*キャッシュロックダウンとキャッシュ保守との相互動作* P. B2-18 で説明されているキャッシュ保守操作の動作が適用されます。この動作は、セキュリティ拡張機能の影響を受けません。

エントリがキャッシュ内でロックされている場合またはロックされる可能性がある場合にアボートを生成する実装で、ロックダウンアボートの使用が許可されていれば、セキュリティ拡張機能にかかわらず、任意のキャッシュ保守操作でこれらのアボートが発生する可能性があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.6 Memory access control

Access to a memory region is controlled by the access permission bits and the domain field in the TLB entry. These form part of the translation table entry formats described in *Translation tables* on page B3-7. The bits and fields are summarized in *First-level descriptors* on page B3-8 and *Second-level descriptors* on page B3-10.

The TLB memory access controls are described in:

- *Access permissions*
- *The Execute Never (XN) attribute and instruction prefetching* on page B3-30
- *Domains* on page B3-31.

B3.6.1 Access permissions

The access permission bits control access to the corresponding memory region. If an access is made to an area of memory without the required permissions, a Permission fault is generated if the domain is set to Client. The access permissions are determined by the AP[2:0] bits in the translation table entry. The XN bit in the translation table entry provides an additional permission bit for instruction fetches.

Note

- Before VMSAv7, the SCTLR.S and SCTLR.R bits also affect the access permissions. For more information, see *Translation attributes* on page AppxH-22.
- From VMSAv7, the full set of access permissions shown in Table B3-4 are only supported when the SCTLR.AFE bit is set to 0. When SCTLR.AFE = 1, the only supported access permissions are those described in *Simplified access permissions model* on page B3-29.
- In previous issues of the *ARM Architecture Reference Manual* and in some other documentation, the AP[2] bit in the translation table entries is described as the APX bit.

Table B3-4 shows the encoding of the access permissions:

Table B3-4 VMSAv7 MMU access permissions

AP[2]	AP[1:0]	Privileged permissions	User permissions	Description
0	00	No access	No access	All accesses generate Permission faults
0	01	Read/write	No access	Privileged access only
0	10	Read/write	Read-only	Writes in User mode generate Permission faults
0	11	Read/write	Read/write	Full access
1	00	-	-	Reserved

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.6 メモリアクセス制御

メモリ領域へのアクセスは、アクセス許可ビットと、TLB エントリのドメインフィールドによって制御されます。これらは、*変換テーブル* P. B3-7 で説明されている変換テーブルエントリのフォーマットの一部を形成します。これらのビットとフィールドについては、*第 1 レベル記述子* P. B3-8 と *第 2 レベル記述子* P. B3-10 で説明します。

TLB メモリアクセスの制御については、次のセクションで説明します。

- *アクセス許可*
- *実行不可 (XN) 属性と命令のプリフェッチ* P. B3-30
- *ドメイン* P. B3-31

B3.6.1 アクセス許可

アクセス許可ビットによって、対応するメモリ領域へのアクセスが制御されます。必要な許可なしにメモリ領域へアクセスを行うと、ドメインがクライアントに設定されている場合はアクセス許可フォールトが生成されます。アクセス許可は、変換テーブルエントリの AP[2:0] ビットによって決定されます。変換テーブルエントリの XN ビットによって、命令フェッチに関する追加の許可ビットが提供されます。

注

- VMSAv7 以前は、SCTLR.S ビットと SCTLR.R ビットもアクセス許可に影響を与えます。詳細については、*変換属性* P. 付録 H-22 を参照して下さい。
- VMSAv7 およびそれ以降では、SCTLR.AFE ビットが 0 にセットされている場合のみ、表 B3-4 に示すアクセス許可の完全なセットがサポートされます。SCTLR.AFE = 1 の場合は、*単純なアクセス許可モデル* P. B3-29 で説明されているアクセス許可のみがサポートされます。
- 旧版の『*ARM アーキテクチャ リファレンスマニュアル*』と他の一部の説明書では、変換テーブルエントリの AP[2] ビットは APX ビットと表記されています。

アクセス許可のエンコードを、表 B3-4 に示します。

表 B3-4 VMSAv7 MMU のアクセス許可

AP[2]	AP[1:0]	特権アクセス許可	ユーザアクセス許可	説明
0	00	アクセス不可	アクセス不可	すべてのアクセスでアクセス許可フォールトが発生
0	01	読み出し / 書き込み	アクセス不可	特権アクセスのみ
0	10	読み出し / 書き込み	読み出し専用	ユーザモードで書き込みを行うとアクセス許可フォールトが発生
0	11	読み出し / 書き込み	読み出し / 書き込み	完全アクセス
1	00	–	–	予約

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-4 VMSAv7 MMU access permissions (continued)

AP[2]	AP[1:0]	Privileged permissions	User permissions	Description
1	01	Read-only	No access	Privileged read-only
1	10	Read-only	Read-only	Privileged and User read-only, deprecated in VMSAv7[a]
1	11	Read-only	Read-only	Privileged and User read-only[b]

a. From VMSAv7, ARM strongly recommends that the 0b11 encoding is used for Privileged and User read-only.
b. This mapping is introduced in VMSAv7, and is reserved in VMSAv6. For more information, see *Simplified access permissions model*.

Each memory region can be tagged as not containing executable code. If the *Execute-never* (XN) bit is set to 1 and the region is in a Client domain, any attempt to execute an instruction in that region results in a Permission fault. If the XN bit is 0 and there is valid read permission, code can execute from that memory region, provided that no other Prefetch Abort condition exists.

Note

The XN bit is ignored on accesses to Manager domains.

Simplified access permissions model

Some memory management require a simple access permissions model where:

- one flag selects between read-only and read/write access
- a second flag selects between User and Kernel control.

In the ARM architecture, this model permits four access combinations:

- read-only by both privileged and unprivileged code
- read/write by both privileged and unprivileged code
- read-only by privileged code, no access by unprivileged code
- read/write by privileged code, no access by unprivileged code.

With the VMSAv7 MMU access permissions shown in Table B3-4 on page B3-28, this model is implemented by:

- Setting the AP[0] bit to 1, unless the SCTLR.AFE bit is set to 1, see *c1, System Control Register (SCTLR)* on page B3-96.
- Using the AP[2:1] bits to control access, as shown in Table B3-5 on page B3-30.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B3-4 VMSAv7 MMU のアクセス許可（続き）

AP[2]	AP[1:0]	特権アクセス許可	ユーザアクセス許可	説明
1	01	読み出し専用	アクセス不可	特権読み出し専用
1	10	読み出し専用	読み出し専用	特権およびユーザ読み出し専用、VMSAv7 では非推奨[a]
1	11	読み出し専用	読み出し専用	特権およびユーザ読み出し専用[b]

a. VMSAv7 およびそれ以降では、特権およびユーザ読み出し専用には 0b11 エンコードを使用することを強くお勧めします。
b. このマッピングは VMSAv7 で導入されたもので、VMSAv6 では予約済みです。詳細については、*単純なアクセス許可モデル*を参照して下さい。

メモリ領域ごとに、実行可能コードが含まれていないことを示すタグを付けることができます。*実行不可*(XN) ビットが 1 にセットされ、その領域がクライアントドメインにある場合、その領域で命令の実行を試みるとアクセス許可フォールトが発生します。XN ビットが 0 で有効な読み出し許可が存在する場合は、他のプリフェッチアボート条件が存在しないかぎり、そのメモリ領域からコードを実行することができます。

注

マネージャドメインへのアクセスでは XN ビットが無視されます。

単純なアクセス許可モデル

メモリ管理によっては、次のような単純なアクセス許可モデルが必要な場合があります。

- 1つのフラグで、読み出し専用アクセスと読み出し / 書き込みアクセスのいずれかを選択する。
- 2 番目のフラグで、ユーザ制御とカーネル制御を選択する。

ARM アーキテクチャでは、このモデルで次に示す 4 種類のアクセスの組み合わせが可能です。

- 特権コードと非特権コードの両方による読み出し専用
- 特権コードと非特権コードの両方による読み出し / 書き込み
- 特権コードによる読み出し専用、非特権コードによるアクセス不可
- 特権コードによる読み出し / 書き込み、非特権コードによるアクセス不可

P. B3-28 の表 B3-4 に示す VMSAv7 MMU のアクセス許可を使用すれば、このモデルは次のように実装されます。

- SCTLR.AFE ビットが 1 にセットされていない場合に AP[0] ビットを 1 にセットする。*c1、システム制御レジスタ (SCTLR)* P. B3-96 を参照して下さい。
- P. B3-30 の表 B3-5 に示すように、AP[2:1] ビットを使用してアクセスを制御する。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-5 VMSAv7 simple access control model

AP[2]	AP[1]	Access[a]
0	0	Kernel, read/write
0	1	User, read/write
1	0	Kernel, read-only
1	1	User, read-only

a. Kernel access corresponds to access by privileged code only.

Note

This model depends on the definition of the AP[2] == 1, AP[1:0] == 0b11 encoding shown in Table B3-4 on page B3-28. This encoding is introduced in VMSAv7, and therefore the simplified access permissions model cannot be supported in VMSAv6.

When the SCTLR.AFE bit is set to 1 the AP[0] bit becomes an access flag, see *The access flag* on page B3-21. In this case, this simplified access permissions model becomes the only supported access permissions model.

B3.6.2 The Execute Never (XN) attribute and instruction prefetching

An implementation must not fetch instructions from any memory location that is marked as Execute Never. A location is marked as Execute Never when it has its XN attribute set to 1 in a Client domain. When the MMU is enabled, instructions can only be fetched or prefetched from memory locations in Client domains where:

- XN is set to 0
- valid read permissions exist
- no other Prefetch Abort condition exists.

Any region of memory that is read-sensitive must be marked as Execute Never, to avoid the possibility of a speculative prefetch accessing the memory region. For example, any memory region that corresponds to a read-sensitive peripheral must be marked as Execute Never.

The XN attribute is not checked for domains marked as Manager. Read-sensitive memory must not be included in domains marked as Manager, because the XN bit does not prevent prefetches in these cases.

The XN attribute is not checked when the MMU is disabled. All VMSAv7 implementations must ensure that, when the MMU is disabled, prefetching down non-sequential paths cannot cause unwanted accesses to read-sensitive devices.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 B3-5 VMSAv7 の単純なアクセス制御モデル

AP[2]	AP[1]	アクセス[a]
0	0	カーネル、読み出し / 書き込み
0	1	ユーザ、読み出し / 書き込み
1	0	カーネル、読み出し専用
1	1	ユーザ、読み出し専用

a. カーネルアクセスは、特権コードのみによるアクセスに相当します。

注

このモデルは、P. B3-28 の表 B3-4 に示す AP[2] == 1 と AP[1:0] == 0b11 のエンコードの定義に依存します。このエンコードは VMSAv7 で導入されたため、VMSAv6 では単純なアクセス許可モデルをサポートできません。

SCTLR.AFE ビットが 1 にセットされている場合は、AP[0] ビットがアクセスフラグになります。アクセスフラグ P. B3-21 を参照して下さい。この場合は、この単純なアクセス許可モデルが、唯一のサポートされるアクセス許可モデルになります。

B3.6.2 実行不可 (XN) 属性と命令のプリフェッチ

実装では、実行不可としてマークされたメモリ位置から命令をフェッチしないようにする必要があります。XN 属性が 1 にセットされたクライアントドメイン内の位置は、実行不可としてマークされます。MMU が許可されている場合は、次のようなクライアントドメイン内のメモリ位置からのみ命令をフェッチまたはプリフェッチできます。

- XN が 0 にセットされている。
- 有効な読み出し許可が存在する。
- 他のプリフェッチアボート条件が存在しない。

メモリ領域への投機的なプリフェッチアクセスの可能性を避けるために、読み出しで影響を受けるメモリ領域は実行不可としてマークする必要があります。例えば、読み出しで影響を受けるペリフェラルに対応するメモリ領域は実行不可としてマークする必要があります。

マネージャとしてマークされたドメインでは XN 属性がチェックされません。したがって、XN ビットによってプリフェッチを阻止できないため、マネージャとしてマークされたドメインには、読み出しで影響を受けるメモリを含めないようにする必要があります。

MMU が禁止されている場合は、XN 属性がチェックされません。すべての VMSAv7 実装において、MMU が禁止されている場合は、非シーケンシャルなパスに沿ったプリフェッチによって、読み出しで影響を受けるデバイスへの不要なアクセスが発生しないことを保証する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.6.3 Domains

A domain is a collection of memory regions. The ARM VMSA architecture supports 16 domains, and each VMSA memory region is assigned to a domain:

- First-level translation table entries for Page tables and Sections include a domain field.
- Translation table entries for Supersections do not include a domain field. Supersections are defined as being in domain 0.
- Second-level translation table entries inherit a domain setting from the parent first-level Page table entry.
- Each TLB entry includes a domain field.

A domain field specifies which domain the entry is in. Access to each domain is controlled by a two-bit field in the Domain Access Control Register, see *c3, Domain Access Control Register (DACR)* on page B3-119. Each field enables the access to an entire domain to be enabled and disabled very quickly, so that whole memory areas can be swapped in and out of virtual memory very efficiently. The VMSA supports two kinds of domain access:

Clients Users of domains, guarded by the access permissions of the TLB entries for that domain. Clients execute programs and access data held in the domain.

Managers Control the behavior of the domain, and are not guarded by the access permissions for TLB entries in that domain. The domain behavior controlled by a Manager covers:
- the sections and pages currently in the domain
- the current access permissions for the domain.

A single program might:
- be a Client of some domains
- be a Manager of some other domains
- have no access to the remaining domains.

This permits very flexible memory protection for programs that access different memory resources. Table B3-6 shows the encoding of the bits in the DACR.

Table B3-6 Domain access values

Value	Access types	Description
00	No access	Any access generates a Domain fault
01	Client	Accesses are checked against the access permission bits in the TLB entry
10	Reserved	Using this value has UNPREDICTABLE results
11	Manager	Accesses are not checked against the access permission bits in the TLB entry, so a Permission fault cannot be generated

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.6.3 ドメイン

ドメインは、メモリ領域の集合です。ARM VMSA アーキテクチャでは、16 個のドメインがサポートされており、各 VMSA メモリ領域がドメインに割り当てられます。

- ページテーブルおよびセクションの第 1 レベル変換テーブルエントリには、ドメインフィールドがあります。
- スーパーセクション用の変換テーブルエントリにはドメインフィールドがありません。スーパーセクションは、ドメイン 0 に存在するものとして定義されます。
- 第 2 レベル変換テーブルエントリは、親の第 1 レベル ページテーブルエントリからドメイン設定を継承します。
- 各 TLB エントリにはドメインフィールドがあります。

ドメインフィールドは、エントリが存在するドメインを指定します。各ドメインへのアクセスは、ドメインアクセス制御レジスタの 2 ビットのフィールドで制御されます。*c3、ドメインアクセス制御レジスタ (DACR)* P. B3-119 を参照して下さい。フィールドごとにドメイン全体へのアクセスの可能 / 不可能を迅速に設定することができるため、非常に効率良く、メモリ領域全体を仮想メモリにスワップインまたはスワップアウトできます。VMSA では、次の 2 種類のドメインアクセスがサポートされています。

クライアント ドメインのユーザで、ドメインの TLB エントリのアクセス許可によって保護されます。クライアントは、プログラムを実行し、ドメイン内に保持されているデータにアクセスします。

マネージャ ドメインの動作を制御しますが、そのドメインの TLB エントリのアクセス許可によって保護されません。マネージャによって制御されるドメインの動作は次の範囲に及びます。
- ドメインに存在するセクションとページ
- 現在のドメインのアクセス許可

単一のプログラムは、次の条件に当てはまる可能性があります。
- 複数のドメインのクライアントである。
- 他の複数のドメインのマネージャである。
- 残りのドメインにアクセスできない。

これによって、複数のメモリリソースにアクセスするプログラムに対して、非常に柔軟なメモリ保護が可能になります。DACR のビットのエンコードを、表 B3-6 に示します。

表 B3-6 ドメインアクセスの値

値	アクセスタイプ	説明
00	アクセス不可	すべてのアクセスでドメインフォールトが発生します。
01	クライアント	アクセスは、TLB エントリのアクセス許可ビットに対してチェックされます。
10	予約	この値の使用による結果は予測不能です。
11	マネージャ	TLB エントリのアクセス許可ビットに対してアクセスがチェックされないため、アクセス許可フォールトが発生しません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.7 Memory region attributes

Each TLB entry has an associated set of memory region attributes. These control accesses to the caches, how the write buffer is used, and if the memory region is Shareable and therefore must be kept coherent. From VMSAv6:

- Most of the memory attributes are controlled by the C and B bits and the TEX[2:0] field of the translation table entries. More information about these attributes is given in the sections:
  - — *The alternative descriptions of the Memory region attributes*
  - — *C, B, and TEX[2:0] encodings without TEX remap* on page B3-33
  - — *Memory region attribute descriptions when TEX remap is enabled* on page B3-34.
- When the Security Extensions are implemented, the NS bit provides an additional memory attribute, see *Secure and Non-secure address spaces* on page B3-26.

Note

The *Bufferable* (B), *Cacheable* (C), and *Type Extension* (TEX) bit names are inherited from earlier versions of the architecture. These names no longer adequately describe the function of the B, C, and TEX bits.

The translation table entries also include an S bit. This bit:

- Is ignored if the entry refers to Device or Strongly-ordered memory.
- For Normal memory, determines whether the memory region is Shareable or Non-shareable:

  S == 0 Normal memory region is Non-shareable

  S == 1 Normal memory region is Shareable.

B3.7.1 The alternative descriptions of the Memory region attributes

From VMSAv7, there are two alternative schemes for describing the memory region attributes, and the current scheme is selected by the SCTLR.TRE (TEX Remap Enable) bit, see *c1, System Control Register (SCTLR)* on page B3-96. The two schemes are:

TRE == 0 TEX Remap disabled. TEX[2:0] are used, with the C and B bits, to describe the memory region attributes.

This is the scheme used in VMSAv6, and it is described in *C, B, and TEX[2:0] encodings without TEX remap* on page B3-33.

TRE == 1 TEX Remap enabled. TEX[2:1] are reassigned for use as flags managed by the operating system. The TEX[0], C and B bits are used to describe the memory region attributes, with the MMU remap registers:

- the Primary Region Remap Register, PRRR
- the Normal Memory Remap Register, NMRR.

This scheme is described in *Memory region attribute descriptions when TEX remap is enabled* on page B3-34.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.7 メモリ領域の属性

各 TLB エントリには、一連のメモリ領域属性が関連付けられます。これらの属性によって、キャッシュに対するアクセス、ライトバッファの使用方法、およびメモリ領域が共有可能かどうかが制御されるため、これらの属性は常に整合している必要があります。VMSAv6 およびそれ以降では、次の規則が適用されます。

- ほとんどのメモリ属性は、C ビット、B ビット、および変換テーブルエントリの TEX[2:0] フィールドによって制御されます。これらの属性の詳細については、次のセクションで説明します。
  - – *メモリ領域の属性の代替表記*
  - – *TEX 再マップを伴わない C、B、TEX[2:0] のエンコード* P. B3-33
  - – *TEX 再マップが許可されている場合のメモリ領域の属性の説明* P. B3-34
- セキュリティ拡張機能が実装されている場合は、NS ビットによって追加のメモリ属性が提供されます。*セキュアアドレス空間と非セキュアアドレス空間* P. B3-26 を参照して下さい。

注

バッファ可能(B)、*キャッシュ可能*(C)、*タイプ拡張*(TEX) の各ビット名は、以前のバージョンのアーキテクチャから継承されたものです。これらの名前は、現在では B、C、TEX ビットの機能を適切に表現しなくなっています。

変換テーブルエントリには S ビットも含まれます。このビットの特徴は次のとおりです。

- エントリがデバイスメモリまたはストロングリオーダメモリを参照している場合は無視されます。
- ノーマルメモリの場合は、メモリ領域が共有可能と共有不可のどちらかを示します。

  S == 0 ノーマルメモリ領域が共有不可です。

  S == 1 ノーマルメモリ領域が共有可能です。

B3.7.1 メモリ領域の属性の代替表記

VMSAv7 から、メモリ領域の属性を示すための 2 つの代替方式が追加され、現在の方式は SCTLR.TRE（TEX 再マップイネーブル）ビットで選択されます。*c1、システム制御レジスタ (SCTLR)* P. B3-96 を参照して下さい。この 2 つの方式は次のとおりです。

TRE == 0 TEX 再マップ禁止。C ビット、B ビットとともに TEX[2:0] を使用してメモリ領域の属性が記述されます。

これは VMSAv6 で使用されている方式で、これについては *TEX 再マップを伴わない C、B、TEX[2:0] のエンコード* P. B3-33 で説明します。

TRE == 1 TEX 再マップ許可。TEX[2:1] が、オペレーティングシステムで管理されるフラグとして再割り当てされます。TEX[0]、C、B ビットは、次の MMU 再マップレジスタとともに、メモリ領域の属性の記述に使用されます。

- 1 次領域再マップレジスタ (PRRR)
- ノーマルメモリ再マップレジスタ (NMRR)

この方式については、*TEX 再マップが許可されている場合のメモリ領域の属性の説明* P. B3-34 で説明します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When the Security Extensions are implemented, the SCTLR.TRE bit is banked between the Secure and Non-secure states.

B3.7.2 C, B, and TEX[2:0] encodings without TEX remap

Table B3-7 shows the C, B, and TEX[2:0] encodings when TEX remap is disabled (TRE == 0).

Table B3-7 TEX, C, and B encodings when TRE == 0

TEX[2:0]	C	B	Description	Memory type	Page Shareable
000	0	0	Strongly-ordered	Strongly-ordered	Shareable
000	0	1	Shareable Device	Device	Shareable
000	1	0	Outer and Inner Write-Through, no Write-Allocate	Normal	S bit[a]
000	1	1	Outer and Inner Write-Back, no Write-Allocate	Normal	S bit[a]
001	0	0	Outer and Inner Non-cacheable	Normal	S bit[a]
001	0	1	Reserved	-	-
001	1	0	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED
001	1	1	Outer and Inner Write-Back, Write-Allocate	Normal	S bit[a]
010	0	0	Non-shareable Device	Device	Non-shareable
010	0	1	Reserved	-	-
010	1	X	Reserved	-	-
011	X	X	Reserved	-	-
1BB	A	A	Cacheable memory: AA = Inner attribute[b] BB = Outer attribute	Normal	S bit[a]

a. Whether the memory is Shareable depends on the value of the S bit, see description in *Memory region attributes* on page B3-32.
b. For more information, see *Cacheable memory attributes* on page B3-34.

See *Memory types and attributes and the memory order model* on page A3-24 for an explanation of Normal, Strongly-ordered and Device memory types and of the Shareable attribute.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

セキュリティ拡張機能が実装されている場合は、SCTLR.TRE ビットがセキュア状態と非セキュア状態との間でバンクされます。

B3.7.2 TEX 再マップを伴わない C、B、TEX[2:0] のエンコード

TEX 再マップが禁止されている場合 (TRE == 0) の C、B、TEX[2:0] のエンコードを、表 B3-7 に示します。

表 B3-7 TRE == 0 の場合の TEX、C、B のエンコード

TEX[2:0]	C	B	説明	メモリタイプ	ページ共有可能
000	0	0	ストロングリオーダ	ストロングリオーダ	共有可能
000	0	1	共有可能デバイス	デバイス	共有可能
000	1	0	外部および内部ライトスルー、書き込み割り当てなし	ノーマル	S ビット[a]
000	1	1	外部および内部ライトバック、書き込み割り当てなし	ノーマル	S ビット[a]
001	0	0	外部および内部キャッシュ不可	ノーマル	S ビット[a]
001	0	1	予約	–	–
001	1	0	実装定義	実装定義	実装定義
001	1	1	外部および内部ライトバック、書き込み割り当て	ノーマル	S ビット[a]
010	0	0	共有不可デバイス	デバイス	共有不可
010	0	1	予約	–	–
010	1	X	予約	–	–
011	X	X	予約	–	–
1BB	A	A	キャッシュ可能メモリ AA = 内部属性[b] BB = 外部属性	ノーマル	S ビット[a]

a. メモリが共有可能かどうかは S ビットの値によって決まります。*メモリ領域の属性* P. B3-32 の説明を参照して下さい。
b. 詳細については、*キャッシュ可能メモリ属性* P. B3-34 を参照して下さい。

ノーマル、ストロングリオーダ、デバイスの各メモリタイプと共有可能属性については、*メモリのタイプおよび属性とメモリオーダ モデル* P. A3-24 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Cacheable memory attributes

When TEX[2] == 1, the translation table entry describes Cacheable memory, and the rest of the encoding defines the Inner and Outer cache attributes:

TEX[1:0] defines the Outer cache attribute

C,B defines the Inner cache attribute

The same encoding is used for the Outer and Inner cache attributes. Table B3-8 shows the encoding.

Table B3-8 Inner and Outer cache attribute encoding

Encoding		Cache attribute
0	0	Non-cacheable
0	1	Write-Back, Write-Allocate
1	0	Write-Through, no Write-Allocate
1	1	Write-Back, no Write-Allocate

B3.7.3 Memory region attribute descriptions when TEX remap is enabled

The VMSAv6 scheme for describing the memory region attributes, described in *C, B, and TEX[2:0] encodings without TEX remap* on page B3-33, uses the TEX[2:0], C and B bits to describe all of the options for Inner and Outer cacheability. However, many system software implementations do not need to use all of these options simultaneously. Instead a smaller subset of attributes can be enabled. This alternative functionality is called TEX remap, and permits software to hold software-interpreted values in the translation tables. When TEX remap is enabled:

- only the TEX[0], C and B bits are used to describe the memory region attributes
- fewer attribute options are available at any time
- the available options are configurable using the PRRR and NMRR registers
- TEX[2:1] are not updated by hardware, see *The OS managed translation table bits* on page B3-38.

When TEX remap is enabled:

- For seven of the eight possible combinations of the TEX[0], C and B bits:
  - — a field in the PRRR defines the corresponding memory region as being Normal, Device or Strongly-ordered memory
  - — a field in the NMRR defines the Inner cache attributes that apply if the PRRR field identifies the region as Normal memory
  - — a second field in the NMRR defines the Outer cache attributes that apply if the PRRR field identifies the region as Normal memory.
- The meaning of the eighth combination for the TEX[0], C and B bits is IMPLEMENTATION DEFINED

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

キャッシュ可能メモリ属性

TEX[2] == 1 の場合は、変換テーブルエントリによってキャッシュ可能メモリが記述され、残りのエンコードで内部と外部のキャッシュ属性が定義されます。

TEX[1:0] 外部キャッシュ属性を定義します。

C、B 内部キャッシュ属性を定義します。

内部と外部のキャッシュ属性では同じエンコードが使用されます。エンコードを表 B3-8 に示します。

表 B3-8 内部と外部のキャッシュ属性のエンコード

エンコード		キャッシュ属性
0	0	キャッシュ不可
0	1	ライトバック、書き込み割り当て
1	0	ライトスルー、書き込み割り当てなし
1	1	ライトバック、書き込み割り当てなし

B3.7.3 TEX 再マップが許可されている場合のメモリ領域の属性の説明

TEX 再マップを伴わない C、B、TEX[2:0] のエンコード P. B3-33 で説明されているのは、VMSAv6 でのメモリ領域の属性を記述する方式で、TEX[2:0]、C、B ビットを使用して、内部と外部のキャッシュ可能性に関するすべてのオプションが記述されます。ただし、多くのシステムソフトウェア実装では、これらのオプションのすべてを同時に使用する必要はありません。代わりに、より小さな属性のサブセットを許可することができます。この代替機能は TEX 再マップと呼ばれており、ソフトウェアで解釈した値を変換テーブルに保持することができます。TEX 再マップが許可されている場合、次の規則が適用されます。

- TEX[0]、C、B ビットのみが、メモリ領域の属性を記述するために使用されます。
- 常に、使用可能な属性オプションが減少します。
- 使用可能なオプションは、PRRR レジスタと NMRR レジスタを使用して構成できます。
- TEX[2:1] はハードウェアで更新されません（*OS で管理される変換テーブルビット* P. B3-38 を参照）。

TEX 再マップが許可されている場合、次の規則が適用されます。

- TEX[0]、C、B ビットの 8 つの可能な組み合わせのうちの 7 つについて、次のようになります。
  - – PRRR のフィールドにより、対応するメモリ領域がノーマル、デバイス、ストロングリオーダのいずれかのメモリとして定義されます。
  - – NMRR のフィールドにより、PRRR フィールドによって領域がノーマルメモリとして識別された場合に適用される内部キャッシュ属性が定義されます。
  - – NMRR 内の 2 番目のフィールドにより、PRRR フィールドによって領域がノーマルメモリとして識別された場合に適用される外部キャッシュ属性が定義されます。
- TEX[0]、C、B ビットの 8 番目の組み合わせの意味は、実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- Four bits in the PRRR permit mapping of the Shareable attribute by defining, for the translation table S bit:
  - — the meaning of S == 0 if the region is identified as Device memory
  - — the meaning of S == 1 if the region is identified as Device memory
  - — the meaning of S == 0 if the region is identified as Normal memory
  - — the meaning of S == 1 if the region is identified as Normal memory.

  In each case, the meaning of the Shareable bit value is that the memory region is one of:
  - — Shareable
  - — Non-shareable.

For each of the possible encodings of the TEX[0], C and B bits in a translation table entry, Table B3-9 shows which fields of the PRRR and NMRR registers describe the memory region attributes.

Table B3-9 TEX, C, and B encodings when TRE == 1

Encoding			Memory type[a]	Cache attributes[a, b]:		Outer Shareable attribute[b]
TEX[0]	C	B		Inner cache	Outer cache	
0	0	0	PRRR[1:0]	NMRR[1:0]	NMRR[17:16]	NOT(PRRR[24])
0	0	1	PRRR[3:2]	NMRR[3:2]	NMRR[19:18]	NOT(PRRR[25])
0	1	0	PRRR[5:4]	NMRR[5:4]	NMRR[21:20]	NOT(PRRR[26])
0	1	1	PRRR[7:6]	NMRR[7:6]	NMRR[23:22]	NOT(PRRR[27])
1	0	0	PRRR[9:8]	NMRR[9:8]	NMRR[25:24]	NOT(PRRR[28])
1	0	1	PRRR[11:10]	NMRR[11:10]	NMRR[27:26]	NOT(PRRR[29])
1	1	0	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED
1	1	1	PRRR[15:14]	NMRR[15:14]	NMRR[31:30]	NOT(PRRR[31])

a. For details of the memory type field encodings see *c10, Primary Region Remap Register (PRRR)* on page B3-143. For details of the cache attribute encodings see Table B3-8 on page B3-34.
b. Only applies if the memory type for the region is mapped as Normal memory and the location is Shareable.

To find the meaning of the value of the S bit in a translation table entry you must:

- use Table B3-9 to find the memory type of the region described by the entry
- if the memory type is Strongly-ordered then the region is Shareable
- if the memory type is not Strongly-ordered then look up the memory type and value of the S bit in Table B3-10 on page B3-36 to find which bit of the PRRR defines the Shareable attribute of the region.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- 変換テーブルの S ビットについて次のものを定義することによって、PRRR の 4 つのビットで共有可能属性のマッピングが可能になります。
  - – 領域がデバイスメモリとして識別された場合の S == 0 の意味
  - – 領域がデバイスメモリとして識別された場合の S == 1 の意味
  - – 領域がノーマルメモリとして識別された場合の S == 0 の意味
  - – 領域がノーマルメモリとして識別された場合の S == 1 の意味

  それぞれのケースで、共有可能ビットの値によって、メモリ領域が次のどちらであるかが示されます。
  - – 共有可能
  - – 共有不可

変換テーブルエントリの TEX[0]、C、B ビットの可能なエンコードのそれぞれについて、PRRR レジスタと NMRR レジスタのどのフィールドがメモリ領域の属性を記述するかを、表 B3-9 に示します。

表 B3-9 TRE == 1 の場合の TEX、C、B のエンコード

エンコード			メモリタイプ[a]	キャッシュ属性[a,b]		外部共有可能属性[b]
TEX[0]	C	B		内部キャッシュ	外部キャッシュ	
0	0	0	PRRR[1:0]	NMRR[1:0]	NMRR[17:16]	NOT(PRRR[24])
0	0	1	PRRR[3:2]	NMRR[3:2]	NMRR[19:18]	NOT(PRRR[25])
0	1	0	PRRR[5:4]	NMRR[5:4]	NMRR[21:20]	NOT(PRRR[26])
0	1	1	PRRR[7:6]	NMRR[7:6]	NMRR[23:22]	NOT(PRRR[27])
1	0	0	PRRR[9:8]	NMRR[9:8]	NMRR[25:24]	NOT(PRRR[28])
1	0	1	PRRR[11:10]	NMRR[11:10]	NMRR[27:26]	NOT(PRRR[29])
1	1	0	実装定義	実装定義	実装定義	実装定義
1	1	1	PRRR[15:14]	NMRR[15:14]	NMRR[31:30]	NOT(PRRR[31])

a. メモリタイプ フィールドのエンコードの詳細については、*c10、1 次領域再マップレジスタ (PRRR)* P. B3-143 を参照して下さい。キャッシュ属性のエンコードの詳細については、P. B3-34 の表 B3-8 を参照して下さい。
b. 領域のメモリタイプがノーマルメモリとしてマッピングされ、その位置が共有可能な場合にのみ適用されます。

変換テーブルエントリの S ビットの値の意味を調べるには、次の作業を実行する必要があります。

- 表 B3-9 を使用して、エントリによって記述される領域のメモリタイプを探します。
- メモリタイプがストロングリオーダの場合、領域は共有可能です。
- メモリタイプがストロングリオーダでない場合は、P. B3-36 の表 B3-10 でメモリタイプと S ビットの値を調べて、PRRR のどのビットによって領域の共有可能属性が定義されるかを特定します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-10 Remapping of the S bit

Memory type	Remapping of Shareable attribute when	
	S = 0	S = 1
Strongly-ordered	Shareable[a]	Shareable[a]
Device	PRRR[16]	PRRR[17]
Normal	PRRR[18]	PRRR[19]

a. No remapping, Strongly-ordered memory is always Shareable.

- The appropriate bit of the PRRR indicates whether the region is Shareable or Non-shareable.

  Note

  When TEX remapping is enabled, it is possible for a translation table entry with S = 0 to be mapped as Shareable memory.

For full descriptions of the TEX remap registers see:

- *c10, Primary Region Remap Register (PRRR)* on page B3-143
- *c10, Normal Memory Remap Register (NMRR)* on page B3-146.

When the Security Extensions are implemented, the TEX remap registers and the SCTLR.TRE bit are banked between the Secure and Non-secure security states. For more information, see *The effect of the Security Extensions on TEX remapping* on page B3-39.

When TEX remap is enabled, the mappings specified by the PRRR and NMRR determine the mapping of the TEX[0], C and B bits in the translation tables to memory type and cacheability attributes:

1. The primary mapping, indicated by a field in the PRRR as shown in the Memory region column of Table B3-9 on page B3-35, takes precedence.
2. Any region that is mapped as Normal memory can have the Inner and Outer Cacheable attributes determined by the NMRR.
3. If it is supported, the Outer Shareable mapping adds a third level of attribute, see *Interpretation of the NOSn fields in the PRRR* on page B3-37.

The TEX remap registers must be static during normal operation. In particular, when the remap registers are changed:

- it is IMPLEMENTATION DEFINED when the changes take effect
- it is UNPREDICTABLE whether the TLB caches the effect of the TEX remap on translation tables.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 B3-10 S ビットの再マッピング

メモリタイプ	共有可能属性の再マッピング	
	S = 0	S = 1
ストロングリオーダ	共有可能[a]	共有可能[a]
デバイス	PRRR[16]	PRRR[17]
ノーマル	PRRR[18]	PRRR[19]

a. 再マッピングなし、ストロングリオーダ メモリは常に共有可能です。

- PRRR の該当するビットによって、領域が共有可能と共有不可のどちらであるかが示されます。

  注

  TEX の再マッピングが許可されている場合は、S = 0 の変換テーブルエントリを共有可能メモリとしてマッピングすることができます。

TEX 再マップレジスタの完全な説明については、次のセクションを参照して下さい。

- *c10、1 次領域再マップレジスタ (PRRR)* P. B3-143
- *c10、ノーマルメモリ再マップレジスタ (NMRR)* P. B3-146

セキュリティ拡張機能が実装されている場合は、TEX 再マップレジスタと SCTLR.TRE ビットがセキュアと非セキュアのセキュリティ状態の間でバンクされます。詳細については、*TEX 再マッピングに対するセキュリティ拡張機能の効果* P. B3-39 を参照して下さい。

TEX 再マップが許可されている場合は、PRRR と NMRR で指定されたマッピングによって、変換テーブルの TEX[0]、C、B ビットのメモリタイプとキャッシュ可能属性へのマッピングが決定されます。

1. P. B3-35 の表 B3-9 のメモリ領域列に示すように、PRRR のフィールドで指定された 1 次マッピングが優先されます。
2. ノーマルメモリとしてマッピングされた領域には、NMRR で決定された内部と外部のキャッシュ可能属性を付与することができます。
3. サポートされていれば、外部共有可能マッピングによって、第 3 レベルの属性が追加されます。*PRRR の NOSn フィールドの解釈* P. B3-37 を参照して下さい。

TEX 再マップレジスタは、通常処理中は静的である必要があります。具体的に、再マップレジスタが変更された場合は、次のようになります。

- 変更が適用されるタイミングは実装定義です。
- 変換テーブルに対する TEX 再マップの効果が TLB でキャッシュされるかどうかは予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The sequence to ensure the synchronization of changes to the TEX remap registers is:

1. Perform a DSB. This ensures any memory accesses using the old mapping have completed.
2. Write the TEX remap registers or SCTLR.TRE bit.
3. Perform an ISB. This ensures synchronization of the register updates.
4. Invalidate the entire TLB.
5. Perform a DSB. This ensures completion of the entire TLB operation.
6. Clean and invalidate all caches. This removes any cached information associated with the old mapping.
7. Perform a DSB. This ensures completion of the cache maintenance.
8. Perform an ISB. This ensures instruction synchronization.

This extends the standard rules for the synchronization of changes to CP15 registers described in *Changes to CP15 registers and the memory order model* on page B3-77, and provides implementation freedom as to whether or not the effect of the TEX remap is cached.

Interpretation of the NOSn fields in the PRRR

When all of the following apply, the NOSn fields in the PRRR distinguish between Inner Shareable and Outer Shareable memory regions:

- the SCTLR.TRE bit is set to 1
- the region is mapped as Normal memory
- the Normal memory remapping of the S bit value for the entry makes the region Shareable
- the implementation supports the distinction between Inner Shareable and Outer Shareable.

If the SCTLR.TRE bit is set to 0, an implementation can provide an IMPLEMENTATION DEFINED mechanism to interpret the NOSn fields in the PRRR, see *SCTLR.TRE, SCTLR.M, and the effect of the MMU remap registers* on page B3-38.

The values of the NOSn fields in the PRRR have no effect if any of the following apply:

- the SCTLR.TRE bit is set to 0 and the IMPLEMENTATION DEFINED mechanism has not been invoked
- the region is not mapped as Normal memory
- the Normal memory remapping of the S bit value for the entry makes the region Non-shareable.

The NOSn fields in the PRRR are RAZ/WI if the implementation does not support the distinction between Inner Shareable and Outer Shareable memory regions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

TEX 再マップレジスタの変更の同期化を保証するためのシーケンスは次のとおりです。

1. DSB を実行します。これによって、古いマッピングを使用したすべてのメモリアクセスが完了していることが保証されます。
2. TEX 再マップレジスタまたは SCTLR.TRE ビットに書き込みます。
3. ISB を実行します。これによって、レジスタ更新の同期化が保証されます。
4. TLB 全体を無効化します。
5. DSB を実行します。これによって、TLB 操作全体の完了が保証されます。
6. すべてのキャッシュをクリーニングおよび無効化します。これによって、古いマッピングに関連付けられた、キャッシュされている情報が削除されます。
7. DSB を実行します。これによって、キャッシュ保守の完了が保証されます。
8. ISB を実行します。これによって、命令の同期化が保証されます。

これによって、*CP15 レジスタの変更とメモリオーダ モデル* P. B3-77 で説明されている CP15 レジスタの変更の同期化に関する標準規則が拡張され、TEX 再マップの効果がキャッシュされるかどうかを実装で決めることができます。

PRRR の NOSn フィールドの解釈

次のすべてが該当する場合は、PRRR の NOSn フィールドによって、内部共有可能メモリ領域と外部共有可能メモリの領域が区別されます。

- SCTLR.TRE ビットが 1 にセットされている。
- 領域がノーマルメモリとしてマッピングされている。
- エントリの S ビット値のノーマルメモリ再マッピングによって領域が共有可能になる。
- 実装で内部共有可能と外部共有可能の区別がサポートされている。

SCTLR.TRE ビットが 0 にセットされている場合は、実装で、PRRR の NOSn フィールドを解釈するための実装定義の機構を提供することができます。*SCTLR.TRE、SCTLR.M、および MMU 再マップレジスタの効果* P. B3-38 を参照して下さい。

次のいずれかが該当する場合は、PRRR の NOSn フィールドの値は意味を持ちません。

- SCTLR.TRE ビットが 0 にセットされ、実装定義の機構が起動されていない。
- 領域がノーマルメモリとしてマッピングされていない。
- エントリの S ビット値のノーマルメモリ再マッピングによって領域が共有不可になる。

実装で内部共有可能と外部共有可能とのメモリ領域の区別がサポートされていない場合、PRRR の NOSn フィールドは RAZ/WI です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

SCTLR.TRE, SCTLR.M, and the effect of the MMU remap registers

When TEX remap is disabled, because the SCTLR.TRE bit is set to 0:

- the effect of the MMU remap registers can be IMPLEMENTATION DEFINED
- the interpretation of the fields of the PRRR and NMRR registers can differ from the description given in this section.

VMSAv7 requires that the effect of these registers is limited to remapping the attributes of memory locations. These registers must not change whether any cache or MMU hardware is enabled. The mechanism by which the MMU remap registers have an effect when the SCTLR.TRE bit is set to 0 is IMPLEMENTATION DEFINED. The ARMv7 architecture requires that from reset, if the IMPLEMENTATION DEFINED mechanism has not been invoked:

- If the MMU is enabled, the architecturally-defined behavior of the TEX[2:0], C, and B bits must apply, without reference to the TEX remap functionality. In other words, memory attribute assignment must comply with the scheme described in *C, B, and TEX[2:0] encodings without TEX remap* on page B3-33.
- If the MMU is disabled, then the architecturally-defined behavior of the VMSA with the MMU disabled must apply, without reference to the TEX remap functionality. See *Enabling and disabling the MMU* on page B3-5.

Typical mechanisms for enabling the IMPLEMENTATION DEFINED effect of the TEX Remap registers when SCTLR.TRE bit is set to 0 include:

- a control bit in the ACTLR, or in a CP15 c15 register
- changing the behavior when the PRRR and NMRR registers are changed from their IMPLEMENTATION DEFINED reset values.

In addition, if the MMU is disabled and the SCTLR.TRE bit is set to 1, the architecturally-defined behavior of the VMSA with the MMU disabled must apply without reference to the TEX remap functionality.

When the Security Extensions are implemented, the IMPLEMENTATION DEFINED effect of these registers must only take effect in the security domain of the registers.

The OS managed translation table bits

When TEX remap is enabled, the TEX[2:1] bits in the translation table descriptors are available as two flags that can be managed by the operating system. In VMSAv7, as long as the SCTLR.TRE bit is set to 1, the values of the TEX[2:1] bits are ignored by the memory management hardware. You can write any value to these bits in the translation tables. In a system that implements access flag updates in hardware, a hardware access flag update never changes these bits.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

SCTLR.TRE、SCTLR.M、および MMU 再マップレジスタの効果

TEX 再マップが禁止されている場合は、SCTLR.TRE ビットが 0 にセットされるため、次の規則が適用されます。

- MMU 再マップレジスタの効果を実装定義にできます。
- PRRR レジスタと NMRR レジスタのフィールドの解釈を、このセクションの説明とは別のものにできます。

VMSAv7 では、これらのレジスタの効果を、メモリ位置の属性の再マッピングに限定する必要があります。キャッシュまたは MMU ハードウェアが許可されているかどうかにかかわらず、これらのレジスタを変更しないようにする必要があります。SCTLR.TRE ビットが 0 にセットされているときに MMU 再マップレジスタが作用する機構は、実装定義です。ARMv7 アーキテクチャでは、リセット後に実装定義の機構が呼び出されていない場合は、次の操作を実行する必要があります。

- MMU が非稼動の場合は、TEX 再マップ機能を参照することなく、TEX[2:0]、C、B ビットのアーキテクチャ定義の動作が適用される必要があります。つまり、メモリ属性の割り当ては、*TEX 再マップを伴わない C、B、TEX[2:0] のエンコード* P. B3-33 で説明されている方式に従う必要があります。
- MMU が禁止されている場合は、TEX 再マップ機能を参照することなく、MMU が非稼動である場合の VMSA のアーキテクチャ定義の動作が適用される必要があります。詳細については、*MMU の許可と禁止* P. B3-5 を参照して下さい。

SCTLR.TRE ビットが 0 にセットされているときに、TEX 再マップレジスタの実装定義の作用を有効にする一般的な機構には、次のものが含まれます。

- ACTLR または CP15 c15 レジスタの制御ビット
- PRRR レジスタと NMRR レジスタが実装定義のリセット時の値から変更された場合の動作の変更

さらに、MMU が非稼動で、SCTLR.TRE ビットが 1 にセットされている場合は、TEX 再マップ機能を参照することなく、MMU が非稼動である場合の VMSA のアーキテクチャ定義の動作が適用される必要があります。

セキュリティ拡張機能が実装されている場合は、これらのレジスタの実装定義の効果が、レジスタのセキュリティドメインにのみ適用される必要があります。

OS で管理される変換テーブルビット

TEX 再マップが可能な場合は、変換テーブル記述子内の TEX[2:1] ビットを、オペレーティングシステムで管理可能な 2 つのフラグとして使用することができます。VMSAv7 では、SCTLR.TRE ビットが 1 にセットされている限り、TEX[2:1] ビットの値はメモリ管理ハードウェアによって無視されます。変換テーブルのこれらのビットには、任意の値を書き込むことができます。ハードウェアによるアクセスフラグの更新が実装されているシステムでは、ハードウェアのアクセスフラグ更新によってこれらのビットが変更されることはありません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.7.4 The effect of the Security Extensions on TEX remapping

When the Security Extensions are implemented, the MMU remap registers are banked in the Secure and Non-secure security states. The register versions for the current security state apply to all TLB lookups. The SCTLR.TRE bit is banked in the Secure and Non-secure copies of the register, and the appropriate version of this bit determines whether TEX remapping is applied to TLB lookups in the current security state.

When the Security Extensions are implemented, the translation table descriptors include an NS bit. For security reasons, the NS bit is not accessible through the MMU remap registers.

Write accesses to the Secure copies of the MMU remap registers are disabled when the **CP15SDISABLE** input is asserted HIGH, and the `MCR` operations to access these registers become UNDEFINED. For more information, see *The CP15SDISABLE input* on page B3-76.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.7.4 TEX 再マッピングに対するセキュリティ拡張機能の効果

セキュリティ拡張機能が実装されている場合は、MMU 再マップレジスタがセキュアと非セキュアのセキュリティ状態の間でバンクされます。現在のセキュリティ状態に対応するバージョンのレジスタが、すべての TLB ルックアップに適用されます。SCTLR.TRE ビットはレジスタのセキュアと非セキュアのコピーでバンクされ、このビットの該当するバージョンによって、現在のセキュリティ状態で TEX 再マッピングが TLB ルックアップに適用されるかどうかが決定されます。

セキュリティ拡張機能が実装されている場合は、変換テーブル記述子に NS ビットが含まれています。セキュリティ上の理由から、NS ビットには、MMU 再マップレジスタ経由でアクセスすることができません。

CP15SDISABLE 入力が HIGH にアサートされ、MMU 再マップレジスタにアクセスするための `MCR` 操作が未定義の場合は、これらのレジスタのセキュアコピーへの書き込みアクセスは不可能です。詳細については、*CP15SDISABLE 入力* P. B3-76 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.8 VMSA memory aborts

The mechanisms that cause the ARM processor to take an exception because of a failed memory access are:

MMU fault The MMU detects an access restriction and signals the processor.

External abort A memory system component other than the MMU signals an illegal or faulting memory access.

The exception taken is a Prefetch Abort exception if either of these occurs synchronously on an instruction fetch, and a Data Abort exception otherwise.

Collectively, these mechanisms are called *aborts*. The different abort mechanisms are described in:

- *MMU faults*
- *External aborts* on page B3-45.

An access that causes an abort is said to be aborted, and uses the *Fault Address Registers* (FARs) and *Fault Status Registers* (FSRs) to record context information. The FARs and FSRs are described in *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48.

Also, a debug exception can cause the processor to take a Prefetch Abort exception or a Data Abort exception, and to update the FARs and FSRs. For details see Chapter C4 *Debug Exceptions* and *Debug event prioritization* on page C3-43.

B3.8.1 MMU faults

The MMU checks the memory accesses required for instruction fetches and for explicit memory accesses:

- if an instruction fetch faults it generates a Prefetch Abort exception
- if an explicit memory access faults it generates a Data Abort exception.

For more information about Prefetch Abort exceptions and Data Abort exceptions see *Exceptions* on page B1-30.

MMU faults are always synchronous. For more information, see *Terminology for describing exceptions* on page B1-4.

When the MMU generates an abort for a region of memory, no memory access is made if that region is or could be marked as Strongly-ordered or Device.

Fault-checking sequence

The sequence used by the MMU to check for access faults is slightly different for sections and pages. For both sections and pages:

- Figure B3-8 on page B3-41 shows the checking sequence
- Figure B3-9 on page B3-42 shows the descriptor fetch and check performed during the checking sequence.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.8 VMSA メモリアボート

メモリアクセスの失敗によって ARM プロセッサで例外が取得される機構は次のとおりです。

MMU フォールト MMU がアクセス制限を検出してプロセッサに通知します。

外部アボート MMU 以外のメモリシステム コンポーネントが、不正なまたは失敗したメモリアクセスを通知します。

取得される例外は、このどちらかが命令フェッチと同期して発生した場合はプリフェッチアボート例外で、それ以外の場合はデータアボート例外です。

これらの機構を総称して*アボート*と呼びます。次のセクションで、さまざまなアボート機構について説明します。

- *MMU フォールト*
- *外部アボート* P. B3-45

アボートを引き起こしたアクセスは、アボートされたと表現され、*フォールトアドレス レジスタ*(FAR) と *フォールトステータス レジスタ*(FSR) を使用してコンテキスト情報が記録されます。FAR と FSR については、*VMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B3-48 で説明します。

また、デバッグ例外によっても、プロセッサでプリフェッチアボート例外またはデータアボート例外が取得され、FAR と FSR が更新される可能性があります。詳細については、第 C4 章 *「デバッグ例外」と デバッグイベントの優先度付け* P. C3-43 を参照して下さい。

B3.8.1 MMU フォールト

MMU では、命令フェッチと明示的なメモリアクセスに必要なメモリアクセスがチェックされます。

- 命令フェッチが失敗した場合は、プリフェッチアボート例外が生成されます。
- 明示的なメモリアクセスが失敗した場合は、データアボート例外が生成されます。

プリフェッチアボート例外とデータアボート例外の詳細については、*例外* P. B1-30 を参照して下さい。

MMU フォールトは必ず同期です。詳細については、*例外に関する用語* P. B1-4 を参照して下さい。

MMU でメモリ領域についてアボートが生成されると、その領域がストロングリオーダまたはデバイスとしてマークされている場合、またはマークされている可能性がある場合はメモリアクセスが実行されません。

フォールトチェック シーケンス

MMU によるアクセスフォールトのチェックに使用されるシーケンスは、セクションやページのシーケンスと少し異なります。セクションおよびページの場合、シーケンスは次のとおりです。

- チェックシーケンスを、P. B3-41 の図 B3-8 に示します。
- チェックシーケンスで実行される記述子のフェッチとチェックを、P. B3-42 の図 B3-9 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure B3-8 VMSA fault checking sequence

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

図 B3-8 VMSA のフォールトチェック シーケンス

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure B3-9 Descriptor fetch and check in the fault checking sequence

The faults that might be detected during the fault checking sequence are described in the following subsections:

- *Alignment fault*
- *External abort on a translation table walk*
- *Translation fault* on page B3-43
- *Access Flag fault* on page B3-43
- *Domain fault* on page B3-44
- *Permission fault* on page B3-44.

Alignment fault

The ARMv7 memory architecture requires support for strict alignment checking. This checking is controlled by the SCTLR.A bit, see *c1, System Control Register (SCTLR)* on page B3-96. For details of when Alignment faults are generated see *Unaligned data access* on page A3-5.

External abort on a translation table walk

This is described in the section *External aborts* on page B3-45, see *External abort on a translation table walk* on page B3-46.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

図 B3-9 フォールトチェック シーケンスでの記述子のフェッチとチェック

フォールトチェック シーケンスで検出される可能性のあるフォールトについては、次のサブセクションで説明します。

- *アライメントフォールト*
- *変換テーブルウォークでの外部アボート*
- *変換フォールト* P. B3-43
- *アクセスフラグ フォールト* P. B3-43
- *ドメインフォールト* P. B3-44
- *アクセス許可フォールト* P. B3-44

アライメントフォールト

ARMv7 メモリアーキテクチャでは、厳密なアライメントチェックをサポートする必要があります。このチェックは SCTLR.A ビットによって制御されます。*c1、システム制御レジスタ (SCTLR)* P. B3-96 を参照して下さい。アライメントフォールトが生成されるタイミングの詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。

変換テーブルウォークでの外部アボート

このアボートについては、*外部アボート* P. B3-45 の *変換テーブルウォークでの外部アボート* P. B3-46 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Translation fault

There are two types of Translation fault:

Section This is generated if the first-level descriptor is marked as invalid. This happens if bits [1:0] of the descriptor are:

- 0b00, the fault encoding
- 0b11, the reserved encoding.

For more information, see *First-level descriptors* on page B3-8.

Page This is generated if the second-level descriptor is marked as invalid. This happens if bits [1:0] of the descriptor are 0b00, the fault encoding. For more information, see *Second-level descriptors* on page B3-10.

Translation table entries that result in Translation faults are guaranteed not to be cached, meaning the TLB is not updated. Therefore, when a Translation fault occurs, it is not necessary to perform any TLB maintenance operations to remove the faulting entries.

Translation faults can be generated by data and unified cache maintenance operations by MVA. It is IMPLEMENTATION DEFINED whether Translation faults can be generated by instruction cache invalidate by MVA operations, see *Instruction cache maintenance operations by MVA* on page B3-25.

It is IMPLEMENTATION DEFINED whether Translation faults can be generated by branch predictor maintenance operations.

Access Flag fault

There are two types of Access Flag fault:

Section This can be generated when a section with AF == 0 is accessed.

Page This can be generated when a page with AF == 0 is accessed.

Access Flag faults only occur on a VMSAv7 implementation that provides software management of the access flag, and are only generated when the AFE flag is set to 1 in the SCTLR, see *c1, System Control Register (SCTLR)* on page B3-96.

Translation table entries that result in Access Flag faults are guaranteed not to be cached, meaning the TLB is not updated. Therefore, when an Access Flag fault occurs, it is not necessary to perform any TLB maintenance operations to remove the faulting entries.

It is IMPLEMENTATION DEFINED whether Access Flag faults can be generated by any cache maintenance operations by MVA.

It is IMPLEMENTATION DEFINED whether Access Flag faults can be generated by branch predictor invalidate by MVA operations.

For more information, see *The access flag* on page B3-21.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

変換フォールト

次の 2 種類の変換フォールトがあります。

セクション これは、第 1 レベル記述子が無効としてマークされた場合に生成されます。記述子のビット [1:0] が次のいずれかの場合に起こります。

- 0b00、フォールトエンコード
- 0b11、予約エンコード

詳細については、*第 1 レベル記述子* P. B3-8 を参照して下さい。

ページ これは、第 2 レベル記述子が無効としてマークされた場合に生成されます。記述子のビット [1:0] が 0b00（フォールトエンコード）の場合に起こります。詳細については、*第 2 レベル記述子* P. B3-10 を参照して下さい。

変換フォールトを引き起こした変換テーブルエントリは、キャッシュされない、つまり TLB が更新されないことが保証されます。したがって、変換フォールトが発生した場合は、TLB 保守操作を実行してフォールトエントリを削除する必要がありません。

変換フォールトは、MVA によるデータおよび統一キャッシュ保守操作によって生成される可能性があります。MVA による命令キャッシュ無効化操作によって変換フォールトが生成されるかどうかは実装定義です。*MVA による命令キャッシュ保守操作* P. B3-25 を参照して下さい。

分岐予測器の保守操作によって変換フォールトが生成されるかどうかは実装定義です。

アクセスフラグ フォールト

次の 2 種類のアクセスフラグ フォールトがあります。

セクション これは、AF = 0 のセクションがアクセスされた場合に生成される可能性があります。

ページ これは、AF = 0 のページがアクセスされた場合に生成される可能性があります。

アクセスフラグ フォールトは、アクセスフラグのソフトウェア管理を提供する VMSAv7 実装でのみ発生し、SCTLR の AFE フラグが 1 にセットされている場合のみ生成されます。*c1、システム制御レジスタ (SCTLR)* P. B3-96 を参照して下さい。

アクセスフラグ フォールトが発生した変換テーブルエントリは、キャッシュされない、つまり TLB が更新されないことが保証されます。したがって、アクセスフラグ フォールトが発生した場合は、TLB 保守操作を実行してフォールトエントリを削除する必要がありません。

MVA によるキャッシュ保守操作によってアクセスフラグ フォールトが生成されるかどうかは実装定義です。

MVA による分岐予測器無効化操作によってアクセスフラグ フォールトが生成されるかどうかは実装定義です。

詳細については、*アクセスフラグ* P. B3-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Domain fault

There are two types of Domain fault:

Section When a first-level descriptor fetch returns a valid Section first-level descriptor, the MMU checks the domain field of that descriptor against the Domain Access Control Register, and generates a Section Domain fault if this check fails.

Page When a second-level descriptor fetch returns a valid second-level descriptor, the MMU checks the domain field of the first-level descriptor that required the second-level fetch against the Domain Access Control Register, and generates a Page Domain fault if this check fails.

Domain faults cannot occur on cache or branch predictor maintenance operations.

For more information, see *Domains* on page B3-31.

Where a Domain fault results in an update to the associated translation tables, the appropriate TLB entry must be flushed to ensure correctness. For more information, see the translation table entry update example in *TLB maintenance operations and the memory order model* on page B3-59.

Changes to the Domain Access Control register must be synchronized by one of:

- performing a ISB operation
- an exception
- exception return.

For details see *Changes to CP15 registers and the memory order model* on page B3-77.

Permission fault

When a memory access is to a Client domain, the MMU checks the access permission field in the translation table entry. As with other MMU faults, there are two types of Permission fault:

Section This can be generated when a section in a Client domain is accessed.

Page This can be generated when a page in a Client domain is accessed.

For details of conditions that cause a Permission fault see *Access permissions* on page B3-28.

Where a Permission fault results in an update to the associated translation tables, the appropriate TLB entry must be flushed to ensure correctness. For more information, see the translation table entry update example in *TLB maintenance operations and the memory order model* on page B3-59.

Permission faults cannot occur on cache or branch predictor maintenance operations.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ドメインフォールト

次の 2 種類のドメインフォールトがあります。

セクション 第 1 レベル記述子のフェッチによって有効なセクション第 1 レベル記述子が返された場合は、MMU が、ドメインアクセス制御レジスタに対して、その記述子のドメインフィールドをチェックし、チェックに失敗した場合にセクションドメイン フォールトを生成します。

ページ 第 2 レベル記述子のフェッチによって有効な第 2 レベル記述子が返された場合は、MMU が、ドメインアクセス制御レジスタに対して、第 2 レベルフェッチが必要な第 1 レベル記述子のドメインフィールドをチェックし、チェックが失敗した場合にページドメイン フォールトを生成します。

ドメインフォールトは、キャッシュまたは分岐予測器の保守操作では発生しません。

詳細については、*ドメイン* P. B3-31 を参照して下さい。

ドメインフォールトによって関連する変換テーブルが更新された場合は、該当する TLB エントリをフラッシュして正確性を保証する必要があります。詳細については、*TLB 保守操作とメモリオーダ モデル* P. B3-59 の変換テーブルエントリの更新例を参照して下さい。

ドメインアクセス制御レジスタの変更は、次のいずれかによって同期する必要があります。

- ISB 操作の実行
- 例外
- 例外からの復帰

詳細については、*CP15 レジスタの変更とメモリオーダ モデル* P. B3-77 を参照して下さい。

アクセス許可フォールト

メモリアクセスがクライアントドメインに対するものである場合は、MMU が、変換テーブルエントリのアクセス許可フィールドをチェックします。他の MMU フォールトと同様に、2 種類のアクセス許可フォールトが存在します。

セクション クライアントドメインのセクションがアクセスされた場合に生成される可能性があります。

ページ クライアントドメインのページがアクセスされた場合に生成される可能性があります。

アクセス許可フォールトを引き起こす条件の詳細については、*アクセス許可* P. B3-28 を参照して下さい。

アクセス許可フォールトによって関連する変換テーブルが更新された場合は、該当する TLB エントリをフラッシュして正確性を保証する必要があります。詳細については、*TLB 保守操作とメモリオーダ モデル* P. B3-59 の変換テーブルエントリの更新例を参照して下さい。

アクセス許可フォールトは、キャッシュまたは分岐予測器の保守操作では発生しません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.8.2 External aborts

External aborts are defined as errors that occur in the memory system other than those that are detected by the MMU or Debug hardware. They include parity errors detected by the caches or other parts of the memory system. An external abort is one of:

- synchronous
- precise asynchronous
- imprecise asynchronous.

For more information, see *Terminology for describing exceptions* on page B1-4.

The ARM architecture does not provide a method to distinguish between precise asynchronous and imprecise asynchronous aborts.

The ARM architecture handles asynchronous aborts in a similar way to interrupts, except that they are reported to the processor using the Data Abort exception. Setting the CPSR.A bit to 1 masks asynchronous aborts, see *Program Status Registers (PSRs)* on page B1-14.

Normally, external aborts are rare. An imprecise asynchronous external abort is likely to be fatal to the process that is running. An example of an event that might cause an external abort is an uncorrectable parity or ECC failure on a Level 2 Memory structure.

It is IMPLEMENTATION DEFINED which external aborts, if any, are supported.

VMSAv7 permits external aborts on data accesses, translation table walks, and instruction fetches to be either synchronous or asynchronous. The DFSR indicates whether the external abort is synchronous or asynchronous, see *c5, Data Fault Status Register (DFSR)* on page B3-121.

Note

Because imprecise external aborts are normally fatal to the process that caused them, ARM recommends that implementations make external aborts precise wherever possible.

More information about possible external aborts is given in the subsections:

- *External abort on instruction fetch* on page B3-46
- *External abort on data read or write* on page B3-46
- *External abort on a translation table walk* on page B3-46
- *Behavior of external aborts on a translation table walk caused by a VA to PA translation* on page B3-46
- *Parity error reporting* on page B3-46.

For details of how external aborts are reported see *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.8.2 外部アボート

外部アボートは、メモリシステムで発生するエラーのうち、MMU またはデバッグハードウェアで検出される以外のものと定義されます。キャッシュまたはメモリシステムの他の部分で検出されたパリティエラーも含まれます。外部アボートには次の種類があります。

- 同期
- 正確な非同期
- 不正確な非同期

詳細については、*例外に関する用語* P. B1-4 を参照して下さい。

ARM アーキテクチャでは、正確な非同期アボートと不正確な非同期アボートとを区別するための方法は提供されません。

ARM アーキテクチャでは、非同期アボートは、データアボート例外を使用してプロセッサに報告されること以外は、割り込みと同様の方法で処理されます。CPSR.A ビットを 1 にセットすると、非同期アボートがマスクされます。*プログラムステータス レジスタ (PSR)* P. B1-14 を参照して下さい。

通常は、外部アボートが発生することはほとんどありません。不正確な非同期外部アボートは、実行中のプロセスにとって致命的な場合があります。外部アボートを引き起こす可能性のあるイベントの例としては、レベル 2 メモリ構造上の修正不可能なパリティや ECC エラーがあります。

外部アボートがある場合、どれをサポートするかは実装定義です。

VMSAv7 では、データアクセス、変換テーブルウォーク、および命令フェッチでの外部アボートを、同期と非同期のどちらにもすることができます。DFSR によって、外部アボートが同期と非同期のどちらかが示されます。*c5、データフォールト ステータスレジスタ (DFSR)* P. B3-121 を参照して下さい。

注

不正確な外部アボートはそれを引き起こしたプロセスにとって一般に致命的なため、実装では外部アボートを可能な限り正確にすることをお勧めします。

可能な外部アボートの詳細については、次のサブセクションで説明します。

- *命令フェッチでの外部アボート* P. B3-46
- *データの読み出し / 書き込みでの外部アボート* P. B3-46
- *変換テーブルウォークでの外部アボート* P. B3-46
- *VA から PA への変換によって引き起こされた変換テーブルウォークでの外部 アボートの動作* P. B3-46
- *パリティエラーの報告* P. B3-47

外部アボートがどのように報告されるかの詳細については、*VMSA 実装のフォールト ステータス レジスタとフォールト アドレス レジスタ* P. B3-48 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

External abort on instruction fetch

An external abort on an instruction fetch can be either synchronous or asynchronous. A synchronous external abort on an instruction fetch is taken precisely.

An implementation can report the external abort asynchronously from the instruction that it applies to. In such an implementation these aborts behave essentially as interrupts. They are masked by the CPSR.A bit when it is set to 1, otherwise they are reported using the Data Abort exception.

External abort on data read or write

Externally generated errors during a data read or write can be either synchronous or asynchronous.

An implementation can report the external abort asynchronously from the instruction that generated the access. In such an implementation these aborts behave essentially as interrupts. They are masked by the CPSR.A bit when it is set to 1, otherwise they are reported using the Data Abort exception.

External abort on a translation table walk

An external abort on a translation table walk can be either synchronous or asynchronous. If the external abort is synchronous then the result is:

- a synchronous Prefetch Abort exception if the translation table walk is for an instruction fetch
- a synchronous Data Abort exception if the translation table walk is for a data access.

An implementation can report the error in the translation table walk asynchronously from executing the instruction whose instruction fetch or memory access caused the translation table walk. In such an implementation these aborts behave essentially as interrupts. They are masked by the CPSR.A bit when it is set to 1, otherwise they are reported using the Data Abort exception.

Behavior of external aborts on a translation table walk caused by a VA to PA translation

The VA to PA translation operations described in *CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130 require translation table walks. An external abort can occur in the translation table walk, as described in *External abort on a translation table walk*. The abort generates a Data Abort exception, and can be synchronous or asynchronous.

Parity error reporting

The ARM architecture supports the reporting of both synchronous and asynchronous parity errors from the cache systems. It is IMPLEMENTATION DEFINED what parity errors in the cache systems, if any, result in synchronous or asynchronous parity errors.

A fault status code is defined for reporting parity errors, see *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48. However when parity error reporting is implemented it is IMPLEMENTATION DEFINED whether the assigned fault status code or another appropriate encoding is used to report parity errors.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

命令フェッチでの外部アボート

命令フェッチでの外部アボートは、同期と非同期のどちらの可能性もあります。命令フェッチでの同期外部アボートは正確に取得されます。

この外部アボートは、実装で、該当する命令とは非同期に報告することができます。このような実装では、これらのアボートが本質的に割り込みとして動作します。この外部アボートは、CPSR.A ビットが 1 にセットされた場合にマスクされ、それ以外の場合はデータアボート例外を使用して報告されます。

データの読み出し / 書き込みでの外部アボート

データの読み出しまたは書き込み時に外部で生成されるエラーは、同期と非同期のどちらの可能性もあります。

この外部アボートは、実装で、アクセスを生成した命令とは非同期に報告することができます。このような実装では、これらのアボートが本質的に割り込みとして動作します。この外部アボートは、CPSR.A ビットが 1 にセットされた場合にマスクされ、それ以外の場合はデータアボート例外を使用して報告されます。

変換テーブルウォークでの外部アボート

変換テーブルウォークでの外部アボートは、同期と非同期のどちらの可能性もあります。この外部アボートが同期の場合は、結果が次のようになります。

- 変換テーブルウォークが命令フェッチに対するものである場合は、同期プリフェッチアボート例外。
- 変換テーブルウォークがデータアクセスに対するものである場合は、同期データアボート例外。

実装で、命令フェッチまたはメモリアクセスによって変換テーブルウォークを引き起こした命令の実行とは非同期に、変換テーブルウォークのエラーを報告することができます。このような実装では、これらのアボートが本質的に割り込みとして動作します。この外部アボートは、CPSR.A ビットが 1 にセットされた場合にマスクされ、それ以外の場合はデータアボート例外を使用して報告されます。

VA から PA への変換によって引き起こされた変換テーブルウォークでの外部アボートの動作

CP15 c7、仮想アドレスから物理アドレスへの変換操作 P. B3-130 で説明されている VA から PA への変換操作では、変換テーブルウォークが必要です。*変換テーブルウォークでの外部アボート*で説明されているように、外部アボートは変換テーブルウォーク時に発生する可能性があります。このアボートは、データアボート例外を生成し、同期と非同期のどちらの可能性もあります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For all purposes other than the fault status encoding, parity errors are treated as external aborts.

B3.8.3 Prioritization of aborts

For synchronous aborts, *Debug event prioritization* on page C3-43 describes the relationship between debug events, MMU faults and external aborts.

In general, the ARM architecture does not define when asynchronous events are taken, and therefore the prioritization of asynchronous events is IMPLEMENTATION DEFINED.

Note

A special requirement applies to asynchronous watchpoints, see *Debug event prioritization* on page C3-43.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

パリティエラーの報告

ARM アーキテクチャでは、キャッシュシステムからの同期と非同期両方のパリティエラーの報告をサポートしています。キャッシュシステムのどのパリティエラーが同期または非同期のパリティエラーになるかは、実装定義です。

パリティエラーを報告するためのフォールトステータス コードが定義されています。*VMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B3-48 を参照して下さい。ただし、パリティエラー報告機能が実装されていても、パリティエラーの報告に、割り当てられたフォールトステータス コードまたは他の適切なエンコードのどちらが使用されるかは実装定義です。

フォールトステータスのエンコード以外のすべての目的には、パリティエラーが外部アボートとして扱われます。

B3.8.3 アボートの優先順位付け

同期アボートについてのデバッグイベント、MMU フォールト、外部アボートの関係については、*デバッグイベントの優先度付け* P. C3-43 で説明します。

通常、ARM アーキテクチャでは、非同期イベントの取得タイミングが定義されていないため、非同期イベントの優先順位付けは実装定義です。

注

非同期ウォッチポイントには、特殊な要件が適用されます。*デバッグイベントの優先度付け* P. C3-43 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.9 Fault Status and Fault Address registers in a VMSA implementation

This section describes the Fault Status and Fault Address registers, and how they report information about VMSA aborts. It contains the following subsections:

- *About the Fault Status and Fault Address registers*
- *Data Abort exceptions* on page B3-49
- *Prefetch Abort exceptions* on page B3-49
- *Fault Status Register encodings for the VMSA* on page B3-50
- *Distinguishing read and write accesses on Data Abort exceptions* on page B3-52
- *Provision for classification of external aborts* on page B3-52
- *The Domain field in the DFSR* on page B3-52
- *Auxiliary Fault Status Registers* on page B3-53.

Also, these registers are used to report information about debug exceptions. For details see *Effects of debug exceptions on CP15 registers and the DBGWFAR* on page C4-4.

B3.9.1 About the Fault Status and Fault Address registers

VMSAv7 provides four registers for reporting fault address and status information:

- The *Data Fault Status Register*, see *c5, Data Fault Status Register (DFSR)* on page B3-121. The DFSR is updated on taking a Data Abort exception.
- The *Instruction Fault Status Register*, see *c5, Instruction Fault Status Register (IFSR)* on page B3-122. The IFSR is updated on taking a Prefetch Abort exception.
- The *Data Fault Address Register*, see *c6, Data Fault Address Register (DFAR)* on page B3-124. In some cases, on taking a synchronous Data Abort exception the DFAR is updated with the faulting address. See *Terminology for describing exceptions* on page B1-4 for a description of synchronous exceptions.
- The *Instruction Fault Address Register*, see *c6, Instruction Fault Address Register (IFAR)* on page B3-125. The IFAR is updated with the faulting address on taking a Prefetch Abort exception.

In addition, the architecture provides encodings for two IMPLEMENTATION DEFINED Auxiliary Fault Status Registers, see *Auxiliary Fault Status Registers* on page B3-53.

Note

- On a Data Abort exception that is generated by an instruction cache maintenance operation, the IFSR is also updated.
- Before ARMv7, the *Data Fault Address Register* (DFAR) was called the *Fault Address Register* (FAR).

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.9 VMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ

このセクションでは、フォールトステータス レジスタとフォールトアドレス レジスタについて、およびこれらのレジスタによって VMSA アボートに関する情報がどのように報告されるかについて説明します。このセクションは、次のサブセクションで構成されています。

- *フォールトステータス レジスタとフォールトアドレス レジスタについて*
- *データアボート例外* P. B3-49
- *プリフェッチアボート例外* P. B3-49
- *VMSA のフォールトステータス レジスタのエンコード* P. B3-50
- *データアボート例外での読み出しアクセスと書き込みアクセスとの区別* P. B3-52
- *外部アボートの分類方法* P. B3-52
- *DFSR のドメインフィールド* P. B3-52
- *補助フォールトステータス レジスタ* P. B3-53

これらのレジスタは、デバッグ例外に関する情報の報告にも使用されます。詳細については、*CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響* P. C4-4 を参照して下さい。

B3.9.1 フォールトステータス レジスタとフォールトアドレス レジスタについて

VMSAv7 では、フォールトアドレスとステータス情報を報告するための 4 つのレジスタが提供されます。

- *データフォールト ステータスレジスタ*（*c5*、*データフォールト ステータスレジスタ (DFSR)* P. B3-121 を参照）。DFSR は、データアボート例外の取得時に更新されます。
- *命令フォールト ステータスレジスタ*（*c5*、*命令フォールト ステータスレジスタ (IFSR)* P. B3-122 を参照）。IFSR は、プリフェッチアボート例外の取得時に更新されます。
- *データフォールト アドレスレジスタ*（*c6*、*データフォールト アドレスレジスタ (DFAR)* P. B3-124 を参照）。同期データアボート例外の取得時に、DFAR がフォールトのアドレスで更新される場合があります。同期例外の説明については、*例外に関する用語* P. B1-4 を参照して下さい。
- *命令フォールト アドレスレジスタ*（*c6*、*命令フォールト アドレスレジスタ (IFAR)* P. B3-125 を参照）。IFAR は、プリフェッチアボート例外の検出時にフォールトのアドレスで更新されます。

さらに、このアーキテクチャでは、2 つの実装定義の補助フォールト ステータスレジスタが提供されます。*補助フォールトステータス レジスタ* P. B3-53 を参照して下さい。

注

- 命令キャッシュ保守操作によって生成されたデータアボート例外で、IFSR も更新されます。
- ARMv7 以前は、*データフォールト アドレスレジスタ*(DFAR) は*フォールト アドレスレジスタ*(FAR) と呼ばれていました。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

On a Watchpoint debug exception, the *Watchpoint Fault Address Register* (DBGWFAR) is used to hold fault information. On a watchpoint access the DBGWFAR is updated with the address of the instruction that generated the Data Abort exception. For more information, see *Watchpoint Fault Address Register (DBGWFAR)* on page C10-28.

B3.9.2 Data Abort exceptions

On taking a Data Abort exception the processor:

- updates the DFSR with a fault status code
- if the Data Abort exception is synchronous:
  - — updates the DFSR with whether the faulted access was a read or a write, and the domain number of the access, if applicable
  - — if the Data Abort exception was not caused by a Watchpoint debug event, updates the DFAR with the MVA that caused the Data Abort exception
  - — if the Data Abort exception was caused by a Watchpoint debug event, the DFAR becomes UNKNOWN
- if the Data Abort exception is asynchronous, the DFAR becomes UNKNOWN.

When the Security Extensions are implemented, the security state of the processor immediately after taking the Data Abort exception determines whether the Secure or Non-secure DFSR and DFAR are updated.

If the Data Abort exception is generated by an instruction cache or branch predictor invalidation by MVA, the DFSR indicates an Instruction Cache Maintenance Operation Fault and the IFSR indicates a Translation or Access Flag fault.

On an access that might have multiple aborts, the MMU fault checking sequence and the prioritization of aborts determine which abort occurs. For more information, see *Fault-checking sequence* on page B3-40 and *Prioritization of aborts* on page B3-47.

B3.9.3 Prefetch Abort exceptions

A Prefetch Abort exception is taken synchronously with the instruction that an abort is reported on. This means:

- If the instruction is executed a Prefetch Abort exception is generated.
- If the instruction fetch is issued but the processor does not attempt to execute the instruction no Prefetch Abort exception is generated for that instruction. For example, if the processor branches round the instruction no Prefetch Abort exception is generated.

On taking a Prefetch Abort exception the processor:

- updates the IFSR with a fault status code
- updates the IFAR with the MVA that caused the Prefetch Abort exception.

When the Security Extensions are implemented, the security state of the processor immediately after taking the Prefetch Abort exception determines whether the Secure or Non-secure DFSR and DFAR are updated.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ウォッチポイントデバッグ例外では、フォールト情報を保持するために*ウォッチポイントフォールトアドレスレジスタ*(DBGWFAR) が使用されます。ウォッチポイントアクセスでは、DBGWFAR が、データアボート例外を生成した命令のアドレスで更新されます。詳細については、*ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10- 28 を参照して下さい。

B3.9.2 データアボート例外

データアボート例外の取得時に、プロセッサは次の処理を実行します。

- フォールトステータス コードで DFSR を更新します。
- データアボート例外が同期の場合、次の処理を行います。
  - – 該当する場合は、失敗したアクセスが読み出しと書き込みのどちらか、およびアクセスのドメイン番号で DFSR を更新します。
  - – ウォッチポイントデバッグ イベントでデータアボート例外が引き起こされなかった場合は、データアボート例外を引き起こした MVA で DFAR を更新します。
  - – ウォッチポイントデバッグ イベントでデータアボート例外が引き起こされた場合は、DFAR は未知の値になります。
- データアボート例外が非同期の場合、DFAR は未知の値になります。

セキュリティ拡張機能が実装されている場合、データアボート例外の取得直後のプロセッサのセキュリティ状態によって、セキュアと非セキュアどちらの DFSR および DFAR が更新されるかが決定されます。

データアボート例外が MVA による命令キャッシュまたは分岐予測器の無効化によって生成された場合は、DFSR によって命令キャッシュ保守操作フォールトが示され、IFSR によって変換フォールトまたはアクセスフラグ フォールトが示されます。

複数のアボートが発生した可能性のあるアクセスでは、MMU フォールトチェック シーケンスとアボートの優先順位付けによって、どのアボートが発生するかが決定されます。詳細については、*フォールトチェック シーケンス* P. B3- 40 と*アボートの優先順位付け* P. B3- 47 を参照して下さい。

B3.9.3 プリフェッチアボート例外

プリフェッチアボート例外は、アボートが報告される命令と同期して取得されます。これは次のことを意味します。

- 命令が実行されると、プリフェッチアボート例外が生成されます。
- 命令フェッチが発行されても、プロセッサがその命令の実行を試みなければ、その命令に対するプリフェッチアボート例外は生成されません。例えば、プロセッサがその命令の前で分岐した場合は、プリフェッチアボート例外が生成されません。

プリフェッチアボート例外の取得時に、プロセッサは次の処理を実行します。

- フォールトステータス コードで IFSR を更新します。
- プリフェッチアボート例外を引き起こした MVA で IFAR を更新します。

セキュリティ拡張機能が実装されている場合は、プリフェッチアボート例外の取得直後のプロセッサのセキュリティ状態によって、セキュアと非セキュアどちらの DFSR および DFAR が更新されるかが決定されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.9.4 Fault Status Register encodings for the VMSA

For the fault status encodings for a VMSA implementation see:

- Table B3-11 for the Instruction Fault Status Register (IFSR) encodings
- Table B3-12 on page B3-51 for the Data Fault Status Register (DFSR) encodings.

Note

In previous ARM documentation, the terms precise and imprecise were used instead of synchronous and asynchronous. For details of the more exact terminology introduced in this manual see *Terminology for describing exceptions* on page B1-4.

Table B3-11 VMSAv7 IFSR encodings

IFSR [10,3:0][a]	Source		IFAR	Notes
01100 01110	Translation table walk synchronous external abort	1st level 2nd level	Valid	-
11100 11110	Translation table walk synchronous parity error	1st level 2nd level	Valid	-
00101 00111	Translation fault	Section Page	Valid	MMU fault
00011[b] 00110	Access Flag fault	Section Page	Valid	MMU fault
01001 01011	Domain fault	Section Page	Valid	MMU fault
01101 01111	Permission fault	Section Page	Valid	MMU fault
00010	Debug event		UNKNOWN	See *Software debug events* on page C3-5
01000	Synchronous external abort		Valid	-
10100	IMPLEMENTATION DEFINED		Valid	Lockdown
11010	IMPLEMENTATION DEFINED		Valid	Coprocessor abort
11001	Memory access synchronous parity error		Valid	-

a. All IFSR[10,3:0] values not listed in this table are reserved.
b. Previously, this encoding was a deprecated encoding for Alignment fault. The extensive changes in the memory model in ARMv7 and VMSAv7 mean there should be no possibility of confusing these two uses.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.9.4 VMSA のフォールトステータス レジスタのエンコード

VMSA 実装のフォールトステータスのエンコードについては、次の表を参照して下さい。

- 表 B3-11（命令フォールト ステータスレジスタ (IFSR) のエンコード）
- P. B3-51 の表 B3-12 （データフォールト ステータスレジスタ (DFSR) のエンコード）

注

以前の ARM 説明書では、同期と非同期の代わりに、正確と不正確という用語が使用されていました。本書で採用された用語の詳細については、*例外に関する用語* P. B1-4 を参照して下さい。

表 B3-11 VMSAv7 IFSR のエンコード

IFSR [10、3:0][a]	ソース		IFAR	注
01100 01110	変換テーブルウォーク同期外部アボート	第 1 レベル 第 2 レベル	有効	–
11100 11110	変換テーブルウォーク同期パリティエラー	第 1 レベル 第 2 レベル	有効	–
00101 00111	変換フォールト	セクション ページ	有効	MMU フォールト
00011[b] 00110	アクセスフラグ フォールト	セクション ページ	有効	MMU フォールト
01001 01011	ドメインフォールト	セクション ページ	有効	MMU フォールト
01101 01111	アクセス許可フォールト	セクション ページ	有効	MMU フォールト
00010	デバッグイベント		未知	*ソフトウェアデバッグ イベント* P. C3-5 を参照
01000	同期外部アボート		有効	–
10100	実装定義		有効	ロックダウン
11010	実装定義		有効	コプロセッサアボート
11001	メモリアクセス同期パリティエラー		有効	–

a. この表に記載されていない IFSR[10、3:0] の値はすべて予約されています。
b. 以前は、このエンコードはアライメントフォールトの非推奨エンコードでした。ARMv7 と VMSAv7 におけるメモリモデルの拡張には、この 2 種類のエンコードを混同しないようにするという目的が含まれています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-12 VMSAv7 DFSR encodings

DFSR [10,3:0][a]	Source		DFAR	Domain	Notes
00001	Alignment fault		Valid	UNKNOWN	MMU fault
00100	Instruction cache maintenance fault		Valid	UNKNOWN	-
01100 01110	Translation table walk synchronous external abort	1st level 2nd level	Valid Valid	UNKNOWN Valid	-
11100 11110	Translation table walk synchronous parity error	1st level 2nd level	Valid Valid	UNKNOWN Valid	-
00101 00111	Translation fault	Section Page	Valid Valid	UNKNOWN Valid	MMU fault
00011[b] 00110	Access Flag fault	Section Page	Valid Valid	UNKNOWN Valid	MMU fault
01001 01011	Domain fault	Section Page	Valid Valid	Valid Valid	MMU fault
01101 01111	Permission fault	Section Page	Valid Valid	Valid Valid	MMU fault
00010	Debug event		UNKNOWN	UNKNOWN	See *Software debug events* on page C3-5
01000	Synchronous external abort		Valid	UNKNOWN	-
10100	IMPLEMENTATION DEFINED		-	-	Lockdown
11010	IMPLEMENTATION DEFINED		-	-	Coprocessor abort
11001	Memory access synchronous parity error		Valid	UNKNOWN	-
10110	Asynchronous external abort[c]		UNKNOWN	UNKNOWN	-
11000	Memory access asynchronous parity error		UNKNOWN	UNKNOWN	Including on translation table walk

a. All DFSR[10,3:0] values not listed in this table are reserved.
b. Previously, this encoding was a deprecated encoding for Alignment fault. The extensive changes in the memory model in ARMv7 and VMSAv7 mean there should be no possibility of confusing these two uses.
c. Including asynchronous data external abort on translation table walk or instruction fetch.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B3-12 VMSAv7 DFSR のエンコード

DFSR [10、3:0][a]	ソース		DFAR	ドメイン	注
00001	アライメントフォールト		有効	未知	MMU フォールト
00100	命令キャッシュ保守フォールト		有効	未知	–
01100 01110	変換テーブルウォーク同期外部アボート	第 1 レベル 第 2 レベル	有効 有効	未知 有効	–
11100 11110	変換テーブルウォーク同期パリティエラー	第 1 レベル 第 2 レベル	有効 有効	未知 有効	–
00101 00111	変換フォールト	セクション ページ	有効 有効	未知 有効	MMU フォールト
00011[b] 00110	アクセスフラグ フォールト	セクション ページ	有効 有効	未知 有効	MMU フォールト
01001 01011	ドメインフォールト	セクション ページ	有効 有効	有効 有効	MMU フォールト
01101 01111	アクセス許可フォールト	セクション ページ	有効 有効	有効 有効	MMU フォールト
00010	デバッグイベント		未知	未知	*ソフトウェアデバッグ イベント* P. C3-5 を参照
01000	同期外部アボート		有効	未知	–
10100	実装定義		–	–	ロックダウン
11010	実装定義		–	–	コプロセッサアボート
11001	メモリアクセス同期パリティエラー		有効	未知	–
10110	非同期外部アボート[c]		未知	未知	–
11000	メモリアクセス非同期パリティエラー		未知	未知	変換テーブルウォーク時を含む

a. この表に記載されていない DFSR[10、3:0] の値はすべて予約されています。
b. 以前は、このエンコードはアライメントフォールトの非推奨エンコードでした。ARMv7 と VMSAv7 におけるメモリモデルの拡張には、この 2 種類のエンコードを混同しないようにするという目的が含まれています。
c. 変換テーブルウォークまたは命令フェッチでの非同期データ外部アボートを含みます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Reserved encodings in the IFSR and DFSR encodings tables

A single encoding is reserved for cache and TLB lockdown faults. The details of these faults and any associated subsidiary registers are IMPLEMENTATION DEFINED.

A single encoding is reserved for aborts associated with coprocessors. The details of these faults are IMPLEMENTATION DEFINED.

B3.9.5 Distinguishing read and write accesses on Data Abort exceptions

On a synchronous Data Abort exception, the DFSR.WnR bit, bit [11] of the register, indicates whether the abort occurred on a read access or on a write access. However, for a fault on a CP15 cache maintenance operation, including a fault on a VA to PA translation operation, this bit always indicates a write access fault.

For a fault generated by an `SWP` or `SWPB` instruction, the WnR bit is 0 if a read to the location would have generated a fault, otherwise it is 1.

B3.9.6 Provision for classification of external aborts

An implementation can use the DFSR.ExT and IFSR.ExT bits to provide more information about external aborts:

- DFSR.ExT can provide an IMPLEMENTATION DEFINED classification of external aborts on data accesses
- IFSR.ExT can provide an IMPLEMENTATION DEFINED classification of external aborts on instruction accesses

For all aborts other than external aborts these bits return a value of 0.

B3.9.7 The Domain field in the DFSR

The DFSR includes a domain field. This has been inherited from previous versions of the VMSA. There is no domain field in the IFSR. The domain field of the DFSR is not valid on watchpoints.

From ARMv7, use of the domain field in the DFSR is deprecated. This field might not be supported in future versions of the ARM architecture. ARM strongly recommends that new software does not use this field.

For both Data Abort exceptions and Prefetch Abort exceptions, software can find the domain information by performing a translation table read for the faulting address and extracting the domain field from the translation table entry.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

IFSR と DFSR のエンコード表で予約済みのエンコード

1 つのエンコードが、キャッシュフォールトと TLB ロックダウンフォールト用に予約されています。これらのフォールトおよび関連する補助レジスタの詳細は、実装定義です。

1 つのエンコードが、コプロセッサに関連付けられたアボート用に予約されています。これらのフォールトの詳細は、実装定義です。

B3.9.5 データアボート例外での読み出しアクセスと書き込みアクセスとの区別

同期データアボート例外では、レジスタの DFSR.WnR ビット（ビット [11]）によって、アボートが読み出しアクセスと書き込みアクセスのどちらで発生したかが示されます。ただし、VA から PA への変換操作など CP15 キャッシュ保守操作でのフォールトの場合は、このビットは常に書き込みアクセスフォールトを示します。

`SWP` 命令または `SWPB` 命令によって生成されたフォールトの場合は、その位置への読み出しによってフォールトが生成される場合は WnR ビットが 0 に、それ以外の場合は WnR ビットが 1 になります。

B3.9.6 外部アボートの分類方法

実装で、DFSR.ExT ビットと IFSR.ExT ビットを使用して、外部アボートの詳細を提供することができます。

- DFSR.ExT を使用すれば、データアクセスでの外部アボートについて実装定義の分類が可能になります。
- IFSR.ExT を使用すれば、命令アクセスでの外部アボートについて実装定義の分類が可能になります。

外部アボート以外のアボートでは、これらのビットは 0 の値を返します。

B3.9.7 DFSR のドメインフィールド

DFSR にはドメインフィールドがあります。このフィールドは、以前のバージョンの VMSA から継承されたものです。IFSR にはドメインフィールドがありません。DFSR のドメインフィールドはウォッチポイントについて有効ではありません。

ARMv7 およびそれ以降では、DFSR のドメインフィールドの使用は非推奨です。このフィールドは、将来のバージョンの ARM アーキテクチャでサポートされない可能性があります。新しいソフトウェアではこのフィールドを使用しないことを強くお勧めします。

データアボート例外とプリフェッチアボート例外では、ソフトウェアでフォールトアドレスに関する変換テーブル読み出しを実行し、その変換テーブルエントリからドメインフィールドを抽出することによって、ドメイン情報を取得できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.9.8 Auxiliary Fault Status Registers

ARMv7 architects two Auxiliary Fault Status Registers:

- the Auxiliary Data Fault Status Register (ADFSR)
- the Auxiliary Instruction Fault Status Register (AIFSR).

These registers enable additional fault status information to be returned:

- The position of these registers is architecturally-defined, but the content and use of the registers is IMPLEMENTATION DEFINED.
- An implementation that does not need to report additional fault information must implement these registers as UNK/SBZ. This ensures that a privileged attempt to access these registers does not cause an Undefined Instruction exception.

An example use of these registers would be to return more information for diagnosing parity errors.

See *c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B3-123 for the architectural details of these registers.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.9.8 補助フォールトステータス レジスタ

ARMv7 では、次の 2 つの補助フォールトステータス レジスタが追加されました。

- 補助データフォールト ステータスレジスタ (ADFSR)
- 補助命令フォールト ステータスレジスタ (AIFSR)

これらのレジスタを使用すれば、追加のフォールトステータス情報を返すことができます。

- これらのレジスタの位置付けはアーキテクチャ定義ですが、レジスタの内容と用途は実装定義です。
- 追加のフォールト情報を報告する必要のない実装では、これらのレジスタを UNK/SBZ として実装する必要があります。これによって、これらのレジスタに対する特権アクセスによって未定義命令例外が発生しないことが保証されます。

これらのレジスタの使用例としては、パリティエラーの診断に関する詳細情報の提供があります。

これらのレジスタのアーキテクチャ上の詳細については、*c5、補助データおよび命令フォールトステータスレジスタ (ADFSR および AIFSR)* P. B3-123 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.10 Translation Lookaside Buffers (TLBs)

Translation Lookaside Buffers (TLBs) are an implementation technique that caches translations or translation table entries. TLBs avoid the requirement for every memory access to perform a translation table lookup. The ARM architecture does not specify the exact form of the TLB structures for any design. In a similar way to the requirements for caches, the architecture only defines certain principles for TLBs:

- The architecture has a concept of an entry locked down in the TLB. The method by which lockdown is achieved is IMPLEMENTATION DEFINED, and an implementation might not support lockdown.
- An unlocked entry in the TLB is not guaranteed to remain in the TLB.
- A locked entry in the TLB is guaranteed to remain in the TLB. However, a locked entry in a TLB might be updated by subsequent updates to the translation tables. Therefore it is not guaranteed to remain incoherent with an entry in the translation table if a change is made to the translation tables.
- A translation table entry that returns a Translation fault or an Access fault is guaranteed not to be held in the TLB. However a translation table entry that returns a Domain fault or a Permission fault might be held in the TLB.
- Any translation table entry that does not return a Translation or Access fault might be allocated to an enabled TLB at any time. The only translation table entries guaranteed not to be held in the TLB are those that return a Translation or Access fault.
- Software can rely on the fact that between disabling and re-enabling the MMU, entries in the TLB have not have been corrupted to give incorrect translations.

B3.10.1 Global and non-global regions in the virtual memory map

The VMSA permits the virtual memory map to be divided into global and non-global regions, distinguished by the nG bit in the translation table descriptors:

nG == 0 The translation is global.

nG == 1 The translation is process specific, meaning it relates to the current ASID, as defined by the CONTEXTIDR.

Each non-global region has an associated *Address Space Identifier* (ASID). These identifiers enable different translation table mappings to co-exist in a caching structure such as a TLB. This means that a new mapping of a non-global memory region can be created without removing previous mappings.

For a symmetric multiprocessor cluster where a single operating system is running on the set of processing elements, ARMv7 requires all ASID values to be assigned uniquely. In other words, each ASID value must have the same meaning to all processing elements in the system.

The use of non-global pages when FCSEIDR[31:25] is not 0b0000000 is UNPREDICTABLE.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.10 変換ルックアサイドバッファ（ＴＬＢ）

変換ルックアサイドバッファ(TLB) は、変換または変換テーブルエントリをキャッシュする実装手法です。TLB では、変換テーブルルックアップを実行するための、すべてのメモリアクセスに関する要件が排除されます。ARM アーキテクチャでは、設計について TLB 構造の詳細な形式が指定されていません。キャッシュに関する要件と同様に、アーキテクチャでは TLB に関する特定の原則のみが定義されています。

- アーキテクチャには、TLB 内でロックダウンされるエントリの概念が含まれています。ロックダウンの実現方法は実装定義で、実装によってはロックダウンがサポートされない場合があります。
- TLB 内のロックされていないエントリが TLB 内に残ることは保証されません。
- TLB 内のロックされたエントリが TLB 内に残ることは保証されます。ただし、TLB 内のロックされたエントリは、以降の変換テーブルの更新により更新される可能性があります。したがって、変換テーブルが変更された場合は、変換テーブル内のエントリと不整合のまま維持されることは保証されません。
- 変換フォールトまたはアクセスフォールトを返す変換テーブルエントリは、TLB 内に保持されないことが保証されます。ただし、ドメインフォールトまたはアクセス許可フォールトを返す変換テーブルエントリは、TLB 内に保持される可能性があります。
- 変換フォールトまたはアクセスフォールトを返さない変換テーブルエントリは、いつでも、有効な TLB に割り当てられる可能性があります。TLB 内に保持されないことが保証されている変換テーブルエントリは、変換フォールトまたはアクセスフォールトを返すもののみです。
- ソフトウェアでは、MMU を非稼動にしてから再度稼動するまでの間に、TLB 内のエントリが破損して不正な変換が行われることはないという事実を前提とできます。

B3.10.1 仮想メモリマップのグローバル領域と非グローバル領域

VMSA では、仮想メモリマップをグローバル領域と非グローバル領域とに分割することができます。これらの領域は、変換テーブル記述子の nG ビットにより区別されます。

nG = 0 変換はグローバルです。

nG = 1 変換はプロセス固有です。つまり、CONTEXTIDR の定義に従って、現在の ASID に関連します。

それぞれの非グローバル領域には、*アドレス空間識別子*(ASID) が関連付けられます。これらの識別子により、TLB などのキャッシュ構造で複数の変換テーブルマッピングを共存させることができます。これは、非グローバルメモリ領域の新しいマッピングを、過去のマッピングを削除せずに作成できることを意味します。

ARMv7 では、一連の処理要素上で単一のオペレーティングシステムが動作する対称型マルチプロセッサクラスタの場合は、すべての ASID 値を一意に割り当てる必要があります。つまり、システム内のすべての処理要素にとって、各 ASID 値が同じ意味になるようにする必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.10.2 TLB matching

A TLB is a hardware caching structure for translation table information. Like other hardware caching structures, it is mostly invisible to software. However, there are some situations where it can become visible. These are associated with coherency problems caused by an update to the translation table that has not been reflected in the TLB. The TLB maintenance operations, described in *TLB maintenance* on page B3-56, enable software to prevent any TLB incoherency becoming a problem.

A particular case where the presence of the TLB can become visible is if the translation table entries that are in use under a particular ASID are changed without suitable invalidation of the TLB. This is an issue regardless of whether or not the translation table entries are global. In some cases, the TLB can hold two mappings for the same address, and this can lead to UNPREDICTABLE behavior

TLB block size

When the TLB is scanned, address matching is performed on bits [31:*N*] of the MVA, where *N* is $\log_2$ of the page size, or block size, for the TLB entry. In VMSAv7, a TLB can store entries based on the following block sizes:

Supersections consist of 16MB blocks of memory, $N = 24$
Sections consist of 1MB blocks of memory, $N = 20$
Large pages consist of 64KB blocks of memory, $N = 16$
Small pages consist of 4KB blocks of memory, $N = 12$.

Supersections, Sections and Large pages are supported to permit mapping of a large region of memory while using only a single entry in a TLB.

B3.10.3 TLB behavior at reset

In ARMv7, there is no requirement that a reset invalidates the TLBs. ARMv7 recognizes that an implementation might require caches, including TLBs, to maintain context over a system reset. Possible reasons for doing so include power management and debug requirements.

For ARMv7:

- All TLBs are disabled at reset.
- An implementation can require the use of a specific TLB *invalidation routine*, to invalidate the TLB arrays before they are enabled after a reset. The exact form of this routine is IMPLEMENTATION DEFINED, but if an invalidation routine is required it must be documented clearly as part of the documentation of the device.

  ARM recommends that if an invalidation routine is required for this purpose, the routine is based on the ARMv7 TLB maintenance operations described in *CP15 c8, TLB maintenance operations* on page B3-138.
- When TLBs that have not been invalidated by some mechanism since reset are enabled, the state of those TLBs is UNPREDICTABLE.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

FCSEIDR[31:25] が 0b0000000 でない場合の非グローバルページの使用は、予測不能です。

B3.10.2 TLB 一致

TLB は、変換テーブル情報のハードウェアキャッシュ構造です。他のハードウェアキャッシュ構造と同様に、ソフトウェアからはほとんど透過的です。ただし、状況によっては可視になることがあります。これは、TLB に反映されていない変換テーブルの更新によって引き起こされたコヒーレンシの問題に関係します。*TLB 保守* P. B3-56 で説明されている TLB 保守操作では、ソフトウェアで TLB の不整合が問題になるのを避けることができます。

TLB の存在が可視になる場合の例は、特定の ASID 下で使用されている変換テーブルエントリが、TLB の適切な無効化を行わずに変更された場合です。これは、変換テーブルエントリがグローバルかどうかに関係しない問題です。場合によっては、同じアドレスについて 2 つのマッピングが TLB に保持されることになり、動作が予測不能になる可能性があります。

TLB ブロックサイズ

TLB のスキャン時に、MVA のビット [31:*N*] に対してアドレス一致が実行されます。ここで、*N* は TLB エントリのページサイズまたはブロックサイズの $\log_2$ です。VMSAv7 では、次のブロックサイズに基づいて TLB にエントリをストアすることができます。

スーパーセクション 16MB のメモリブロック ($N = 24$) で構成されます。
セクション 1MB のメモリブロック ($N = 20$) で構成されます。
ラージページ 64KB のメモリブロック ($N = 16$) で構成されます。
スモールページ 4KB のメモリブロック ($N = 12$) で構成されます。

TLB の単一のエントリのみを使用して大きなメモリ領域のマッピングを可能にするため、スーパーセクション、セクション、ラージページがサポートされています。

B3.10.3 リセット時の TLB の動作

ARMv7 では、リセットによって TLB を無効化する要件はありません。また、実装では、システムリセットの前後で、TLB を含むキャッシュがコンテキストを維持することが必要な場合もあります。この動作が必要となる理由には、電力管理とデバッグの要件が含まれます。

ARMv7 では、次の規則が適用されます。

- すべての TLB はリセット時に非稼動になります。
- 実装では、特定の TLB *無効化ルーチン*を使用して、リセット後に TLB アレイを稼動する前に無効化することが必要な場合があります。このルーチンの詳細な形式は実装定義ですが、無効化ルーチンが必要な場合は、デバイスのドキュメントの一部として明確に文書化する必要があります。

  この目的の無効化ルーチンが必要な場合は、*CP15 c8*、*TLB 保守操作* P. B3-138 で説明されている ARMv7 の TLB 保守操作に基づくことをお勧めします。
- リセット後に何らかの機構によって無効化されなかった TLB が稼動された場合、その TLB の状態は予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Similar rules apply:

- to cache behavior, see *Behavior of the caches at reset* on page B2-6
- to branch predictor behavior, see *Behavior of the branch predictors at reset* on page B2-21.

B3.10.4 TLB lockdown

ARMv7 recognizes that any TLB lockdown scheme is heavily dependent on the microarchitecture, making it inappropriate to define a common mechanism across all implementations. This means that:

- ARMv7 does not require TLB lockdown support.
- If TLB lockdown support is implemented, the lockdown mechanism is IMPLEMENTATION DEFINED. However, key properties of the interaction of lockdown with the architecture must be documented as part of the implementation documentation.

This means that:

- In ARMv7, the TLB Type Register TLBTR does not define the lockdown scheme in use. This is a change from previous versions of the architecture.
- A region of the CP15 c10 encodings is reserved for IMPLEMENTATION DEFINED TLB functions, such as TLB lockdown functions. The reserved encodings are those with:
  - — `<CRm>` = {0, 1, 4, 8}
  - — all values of `<opc2>` and `<opc1>`.

  See also *The implementation defined TLB control operations* on page B3-143.

An implementation might use some of the CP15 c10 encodings that are reserved for IMPLEMENTATION DEFINED TLB functions to implement additional TLB control functions. These functions might include:

- Unlock all locked TLB entries.
- Preload into a specific level of TLB. This is beyond the scope of the `PLI` and `PLD` hint instructions.

B3.10.5 TLB maintenance

TLB maintenance operations provide a mechanism to invalidate entries from a TLB.

Any TLB operation might affect other TLB entries that are not locked down.

TLB maintenance operations are provided by CP15 c8 functions. The following operations are supported:

- invalidate all unlocked entries in the TLB
- invalidate a single TLB entry, by MVA, or MVA and ASID for a non-global entry
- invalidate all TLB entries that match a specified ASID.

The Multiprocessing Extensions add the following operations:

- invalidate all TLB entries that match a specified by MVA, regardless of the ASID
- operations that apply across multiprocessors in the same Inner Shareable domain, see *Multiprocessor effects on TLB maintenance operations* on page B3-62.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

次の動作にも、同様の規則が適用されます。

- キャッシュの動作。*リセット時のキャッシュの動作* P. B2-6 を参照して下さい。
- 分岐予測器の動作。*リセット時の分岐予測器の動作* P. B2-21 を参照して下さい。

B3.10.4 TLB ロックダウン

ARMv7 では、TLB ロックダウン方式がマイクロアーキテクチャに大きく依存しているため、すべての実装に共通する機構の定義は不適切です。これは次のことを意味します。

- ARMv7 では、TLB ロックダウンをサポートする必要がありません。
- TLB ロックダウンのサポートを実装する場合、ロックダウン機構は実装定義です。ただし、ロックダウンとアーキテクチャとの相互動作の主要なプロパティを、実装の説明書に記載する必要があります。

これは次のことを意味します。

- ARMv7 では、使用されているロックダウン方式は、TLB タイプレジスタ (TLBTR) によって定義されません。これは、以前のバージョンのアーキテクチャから変更されたものです。
- CP15 c10 エンコードの領域が、TLB ロックダウン関数などの実装定義の TLB 関数用に予約されています。この予約済みのエンコードは、次の条件に該当するものです。
  - — `<CRm>` = {0, 1, 4, 8}
  - — `<opc2>` と `<opc1>` のすべての値

  実装定義の TLB 制御操作 P. B3-143 も参照して下さい。

実装では、実装定義の TLB 関数用に予約された CP15 c10 エンコードの一部を使用して、新しい TLB 制御関数を実装することができます。この関数には次のものを含めることができます。

- ロックされたすべての TLB エントリをロック解除する。
- 特定のレベルの TLB にプリロードする。これは、`PLI` と `PLD` のヒント命令の範囲を超えています。

B3.10.5 TLB 保守

TLB 保守操作では、TLB のエントリを無効化する機構が提供されます。

すべての TLB 操作は、ロックダウンされていない他の TLB エントリに影響を与える可能性があります。

TLB 保守操作は、CP15 c8 関数によって提供されます。次の操作がサポートされています。

- TLB の、ロックされていないすべてのエントリを無効化する。
- MVA、または MVA と非グローバルエントリの ASID によって、単一の TLB エントリを無効化する。
- 指定された ASID と一致するすべての TLB エントリを無効化する。

マルチプロセッシング拡張機能では、次の操作が追加されます。

- ASID にかかわらず、MVA によって指定されたものと一致するすべての TLB エントリを無効化する。
- 同じ内部共有可能ドメイン内のマルチプロセッサ全体に適用される操作（*TLB 保守操作に対するマルチプロセッサの影響* P. B3-62 を参照）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In the TLB operations:

- An operation that depends on an MVA value includes a field for the ASID to be used as part of the translation. For a translation table entry that refers to a non-global region, the ASID must be specified.
- If the Security Extensions are implemented, operations include the current security state as part of the VA to PA address translation required for the TLB operation.

A single register function can apply one of these operations:

- when separate Instruction and Data TLBs are implemented, to:
  - — only the Instruction TLB
  - — only the Data TLB
  - — both the Instruction TLB and the Data TLB
- the Unified TLB, when a Unified TLB is implemented.

The distinction between the Instruction TLB and Data TLB in TLB maintenance operations is historical and is not supported in newer instructions. The distinction is deprecated in ARMv7. Developers must not rely on this distinction being maintained in future versions of the ARM architecture.

The ARM architecture does not dictate the form in which the TLB stores translation table entries. However, for TLB invalidate operations, the size of the table entry that must be removed from the TLB must be at least the size that appears in the translation table entry.

These operations are described in *CP15 c8, TLB maintenance operations* on page B3-138.

The interaction of TLB maintenance operations with TLB lockdown

The precise interaction of TLB lockdown with the TLB maintenance operations is IMPLEMENTATION DEFINED. However, the architecturally-defined TLB maintenance operations must comply with these rules:

- The effect on locked entries of the TLB invalidate all unlocked entries and TLB invalidate by MVA all ASID operations is IMPLEMENTATION DEFINED. However, these operations must implement one of the following options:
  - — Have no effect on entries that are locked down.
  - — Generate an IMPLEMENTATION DEFINED Data Abort exception if an entry is locked down, or might be locked down. A fault status code is provided in the CP15 c5 fault status registers for cache and TLB lockdown faults, see Table B3-11 on page B3-50 and Table B3-12 on page B3-51.

  This permits a typical usage model for TLB invalidate routines, where the routine invalidates a large range of addresses, without considering whether any entries are locked in the TLB.
- The effect on locked entries of the TLB invalidate by MVA and invalidate by ASID match operations is IMPLEMENTATION DEFINED. However, these operations must implement one of these options:
  - — A locked entry is invalidated in the TLB.
  - — The operation has no effect on a locked entry in the TLB. In the case of the Invalidate single entry by MVA, this means the operation is treated as a NOP.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

TLB 操作には、次の規則が適用されます。

- MVA 値に依存する操作には、変換の一部として使用される ASID に関するフィールドが含まれます。非グローバル領域を参照する変換テーブルエントリについては、ASID を指定する必要があります。
- セキュリティ拡張機能が実装されている場合は、TLB 操作に必要な VA から PA へのアドレス変換の一部として、現在のセキュリティ状態が操作に含まれます。

単一レジスタの関数は、次のいずれかの操作に適用できます。

- 独立した命令 TLB とデータ TLB が実装されている場合
  - – 命令 TLB のみ
  - – データ TLB のみ
  - – 命令 TLB とデータ TLB の両方
- 統一 TLB（統一 TLB が実装されている場合）

TLB 保守操作における命令 TLB とデータ TLB の区別は過去に行われていたもので、新しい命令ではサポートされていません。ARMv7 では、この区別は非推奨です。開発者は、この区別が将来のバージョンの ARM アーキテクチャで維持されることを前提としない必要があります。

ARM アーキテクチャでは、変換テーブルエントリを TLB にストアする形式が指定されていません。ただし、TLB 無効化操作の場合は、TLB から削除されるテーブルエントリのサイズを、変換テーブルエントリ内でのサイズ以上にする必要があります。

これらの操作については、*CP15 c8、TLB 保守操作* P. B3-138 で説明します。

TLB 保守操作と TLB ロックダウンとの相互動作

TLB ロックダウンと TLB 保守操作の正確な相互動作は、実装定義です。ただし、アーキテクチャ定義の TLB 保守操作は、次の規則に従う必要があります。

- TLB のロックされていないすべてのエントリを無効化する操作と、MVA とすべての ASID による TLB 無効化操作の、ロックされたエントリに対する作用は、実装定義です。ただし、これらの操作では、次のオプションのいずれかを実装する必要があります。
  - – ロックダウンされたエントリに影響を与えない。
  - – エントリがロックダウンされている場合、または、ロックダウンされている可能性がある場合は、実装定義のデータアボート例外を生成する。キャッシュフォールトと TLB ロックダウンフォールトの場合は、CP15 c5 フォールト ステータス レジスタにフォールト ステータス コードが提供されます（P. B3-50 の表 B3-11 と P. B3-51 の表 B3-12 を参照）。

  これによって、TLB のエントリがロックされているかどうかを考慮する必要なしに、アドレスの広い範囲を無効化する、TLB 無効化ルーチンの一般的な使用モデルが可能になります。
- MVA による TLB 無効化操作と、ASID 一致による無効化操作の、ロックされたエントリに対する作用は、実装定義です。ただし、これらの操作では、次のオプションのいずれかを実装する必要があります。
  - – TLB のロックされたエントリが無効化される。
  - – 操作は、TLB のロックされたエントリに作用しない。MVA による単一エントリの無効化の場合、この操作が NOP として扱われることを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

— The operation generates an IMPLEMENTATION DEFINED Data Abort exception if it operates on an entry that is locked down, or might be locked down. A fault status code is provided in the CP15 c5 fault status registers for cache and TLB lockdown faults, see Table B3-11 on page B3-50 and Table B3-12 on page B3-51.

Any implementation that uses an abort mechanism for entries that might be locked must:

- document the IMPLEMENTATION DEFINED code sequences that then performs the required operations on entries that are not locked down
- implement one of the other specified alternatives for the locked entries.

ARM recommends that architecturally-defined operations are used wherever possible in such sequences, to minimize the number of customized operations required.

In addition, if an implementation uses an abort mechanisms for entries that might be locked it must also must provide a mechanism that ensures that no TLB entries are locked.

Similar rules apply to cache lockdown, see *The interaction of cache lockdown with cache maintenance* on page B2-18.

An unlocked entry in the TLB is not guaranteed to remain in the TLB. This means that, as a side effect of a TLB maintenance operation, any unlocked entry in the TLB might be invalidated.

The effect of the Security Extensions on the TLB maintenance operations

If an implementation includes the Security Extensions, the TLB maintenance operations must take account of the current security state. Table B3-13 summarizes how the Security Extensions affect these operations.

Table B3-13 TLB maintenance operations when the Security Extensions are implemented

TLB maintenance operation	TLB entries guaranteed to be invalidated
Invalidate all entries	All TLB entries accessible in the current security state.
Invalidate single entry by MVA	Targeted TLB entry, only if all of these apply: • the MVA value matches • the ASID value matches, for a non-global entry • the entry applies to the current security state.
Invalidate entries by ASID match	All non-global TLB entries for which both: • the ASID value matches • the entry applies to the current security state.
Invalidate entries by MVA, all ASID	All targeted TLB entries for which both: • the MVA value matches • the entry applies to the current security state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

– ロックダウンされているか、ロックダウンされている可能性があるエントリに対して操作が行われた場合、実装定義のデータアボート例外を生成する。キャッシュフォールトと TLB ロックダウンフォールトの場合は、CP15 c5 フォールトステータスレジスタにフォールトステータスコードが提供されます（P. B3-50 の表 B3-11 と P. B3-51 の表 B3-12 を参照）。

ロックされている可能性のあるエントリについてアボート機構を使用する実装では、次の作業が必要です。

- ロックダウンされていないエントリに対して必要な操作を実行する、実装定義のコードシーケンスを文書化する。
- ロックされたエントリについて指定された代替手段の 1 つを実装する。

このようなシーケンスでは、可能な限りアーキテクチャ定義の操作を使用し、カスタマイズが必要な操作の数を最小限に抑えることをお勧めします。

さらに、ロックされている可能性のあるエントリに対するアボート機構を実装で使用する場合は、ロックされている TLB エントリが存在しないことを保証する機構も提供する必要があります。

キャッシュロックダウンにも同様の規則が適用されます。*キャッシュロックダウンとキャッシュ保守との相互動作* P. B2-18 を参照して下さい。

TLB のロックされていないエントリが TLB に残ることは保証されません。これは、TLB 保守操作の副作用として、TLB のロックされていないエントリが無効にされる可能性があることを意味します。

TLB 保守操作に対するセキュリティ拡張機能の影響

実装にセキュリティ拡張機能が含まれている場合は、TLB 保守操作で現在のセキュリティ状態を考慮する必要があります。表 B3-13 は、セキュリティ拡張機能がこれらの操作にどのように影響するかをまとめたものです。

表 B3-13 セキュリティ拡張機能が実装されている場合の TLB 保守操作

TLB 保守操作	無効化が保証される TLB エントリ
すべてのエントリの無効化	現在のセキュリティ状態でアクセス可能なすべての TLB エントリ
MVA による単一エントリの無効化	次のすべてが該当する場合のみ、ターゲットの TLB エントリ ・ MVA 値が一致する。 ・ ASID 値が一致する（非グローバルエントリの場合）。 ・ エントリが現在のセキュリティ状態に該当する。
ASID 一致によるエントリの無効化	次の両方に該当するすべての非グローバル TLB エントリ ・ ASID 値が一致する。 ・ エントリが現在のセキュリティ状態に該当する。
MVA とすべての ASID によるエントリの無効化	次の両方に該当するすべてのターゲットの TLB エントリ ・ MVA 値が一致する。 ・ エントリが現在のセキュリティ状態に該当する。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The Security Extensions do not change the possible effects of TLB maintenance operations on entries that are locked or might be locked, as described in *The interaction of TLB maintenance operations with TLB lockdown* on page B3-57. If an implementation has TLB maintenance operations that generate aborts on entries that are locked or might be locked then those aborts can occur on any maintenance operation, regardless of the Security Extensions. However aborts must not be generated as a result of entries from the other security state.

TLB maintenance operations and the memory order model

The following rules describe the relations between the memory order model and the TLB maintenance operations:

- A TLB invalidate operation is complete when all memory accesses using the TLB entries that have been invalidated have been observed by all observers to the extent that those accesses are required to be observed, as determined by the shareability and cacheability of the memory locations accessed by the accesses. In addition, once the TLB invalidate operation is complete, no new memory accesses that can be observed by those observers using those TLB entries will be performed.
- A TLB maintenance operation is only guaranteed to be complete after the execution of a `DSB` instruction.
- An `ISB` instruction, or a return from an exception, causes the effect of all completed TLB maintenance operations that appear in program order before the `ISB` or return from exception to be visible to all subsequent instructions, including the instruction fetches for those instructions.
- An exception causes all completed TLB maintenance operations that appear in the instruction stream before the point where the exception was taken to be visible to all subsequent instructions, including the instruction fetches for those instructions.
- All TLB Maintenance operations are executed in program order relative to each other.
- The execution of a Data or Unified TLB maintenance operation is guaranteed not to affect any explicit memory access of any instruction that appears in program order before the TLB maintenance operation. This means no memory barrier instruction is required. This ordering is guaranteed by the hardware implementation.
- The execution of a Data or Unified TLB maintenance operation is only guaranteed to be visible to a subsequent explicit load or store operation after both:
  - — the execution of a `DSB` instruction to ensure the completion of the TLB operation
  - — a subsequent `ISB` instruction, or taking an exception, or returning from an exception.
- The execution of an Instruction or Unified TLB maintenance operation is only guaranteed to be visible to a subsequent instruction fetch after both:
  - — the execution of a `DSB` instruction to ensure the completion of the TLB operation
  - — a subsequent `ISB` instruction, or taking an exception, or returning from an exception.

The following rules apply when writing translation table entries. They ensure that the updated entries are visible to subsequent accesses and cache maintenance operations.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

TLB 保守操作と TLB ロックダウンとの相互動作 P. B3-57 で説明されているように、ロックされている、またはロックされている可能性のあるエントリに対して行われる可能性がある TLB 保守操作の影響は、セキュリティ拡張機能により変更されません。実装に、ロックされている、またはロックされている可能性のあるエントリに対してアボートを生成する TLB 保守操作が含まれている場合は、セキュリティ拡張機能にかかわらず、任意の保守操作でこれらのアボートが発生する可能性があります。ただし、他のセキュリティ状態からの遷移の結果としてアボートが発生しないようにする必要があります。

TLB 保守操作とメモリオーダ モデル

次の規則は、メモリオーダ モデルと TLB 保守操作との関係を示すものです。

- TLB 無効化操作は、無効化された TLB エントリを使用するすべてのメモリアクセスが、アクセス対象のメモリ位置の共有可能性とキャッシュ可能性に従う決定により、そのアクセスが観測される必要がある範囲において、すべてのオブザーバから観測されたときに完了します。また、TLB 無効化操作が完了すると、オブザーバがそれらの TLB エントリを使用して観測可能な、新しいメモリアクセスは実行されなくなります。
- TLB 保守操作は、`DSB` 命令の実行後にのみ、完了が保証されます。
- `ISB` 命令または例外からの復帰が行われると、プログラムの順序で `ISB` 命令または例外からの復帰の前に出現する、すべての完了した TLB 保守操作の効果は、以後のすべての命令（それらの命令のフェッチも含む）から可視になります。
- 例外が発生すると、命令ストリームにおいて、例外が取得された時点よりも前に出現するすべての完了した TLB 保守操作は、以後のすべての命令（それらの命令のフェッチも含む）から可視になります。
- すべての TLB 保守操作は、相対的にプログラム上の順序で実行されます。
- データまたは統一 TLB 保守操作の実行は、プログラムの順序で TLB 保守操作の前に出現する命令の明示的なメモリアクセスに影響を与えないことが保証されます。これは、メモリバリア命令が必要ないことを意味します。この順序はハードウェア実装によって保証されます。
- データまたは統一 TLB 保守操作の実行は、次の両方が行われた後でのみ、以後の明示的なロードまたはストア操作から可視であることが保証されます。
  - – TLB 操作の完了を保証するための `DSB` 命令の実行
  - – 後続の `ISB` 命令、例外の取得、または例外からの復帰
- 命令または統一 TLB 保守操作の実行は、次の両方が行われた後でのみ、以後の命令フェッチから可視になることが保証されます。
  - – TLB 操作の完了を保証するための `DSB` 命令の実行
  - – 後続の `ISB` 命令、例外の取得、または例外からの復帰

変換テーブルエントリの書き込み時には、次の規則が適用されます。これらの規則によって、更新されたエントリは、以後のアクセスおよびキャッシュ保守操作から可視になることが保証されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For TLB maintenance, the translation table walk is treated as a separate observer:

- A write to the translation tables, after it has been cleaned from the cache if appropriate, is only guaranteed to be seen by a translation table walk caused by an explicit load or store after the execution of both a DSB and an ISB.

  However, it is guaranteed that any writes to the translation tables are not seen by any explicit memory access that occurs in program order before the write to the translation tables.

- For the base ARMv7 architecture and versions of the architecture before ARMv7, if the translation tables are held in Write-Back Cacheable memory, the caches must be cleaned to the point of unification after writing to the translation tables and before the DSB instruction. This ensures that the updated translation table are visible to a hardware translation table walk.

- A write to the translation tables, after it has been cleaned from the cache if appropriate, is only guaranteed to be seen by a translation table walk caused by the instruction fetch of an instruction that follows the write to the translation tables after both a DSB and an ISB.

Therefore, typical code for writing a translation table entry, covering changes to the instruction or data mappings in a uniprocessor system is:

```
STR rx, [Translation table entry]         ; write new entry to the translation table
Clean cache line [Translation table entry] : This operation is not required with the
                                           ; Multiprocessing Extensions.
DSB           ; ensures visibility of the data cleaned from the D Cache
Invalidate TLB entry by MVA (and ASID if non-global) [page address]
Invalidate BTC
DSB           ; ensure completion of the Invalidate TLB operation
ISB           ; ensure table changes visible to instruction fetch
```

Synchronization of changes of ASID and TTBR

A common virtual memory management requirement is to change the ContextID and Translation Table Base Registers together to associate the new ContextID with different translation tables. However, such a change is complicated by:

- the depth of prefetch being IMPLEMENTATION DEFINED
- the use of branch prediction.

The virtual memory management operations must ensure the synchronization of changes of the ContextID and the translation table registers. For example, some or all of the TLBs, BTCs (*Branch Target Caches*) and other caching of ASID and translation information might become corrupt with invalid translations. Synchronization is necessary to avoid either:

- the *old* ASID being associated with translation table walks from the *new* translation tables
- the *new* ASID being associated with translation table walks from the *old* translation tables.

There are a number of possible solutions to this problem, and the most appropriate approach depends on the system. Example B3-2 on page B3-61 and Example B3-3 on page B3-61, and Example B3-4 on page B3-62 describe three possible approaches.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

TLB 保守では、変換テーブルウォークが別のオブザーバとして扱われます。

- 変換テーブルへの書き込みは、必要に応じてキャッシュからクリーニングされた後で、DSB と ISB の両方が実行された後の明示的なロードまたはストアにより引き起こされる変換テーブルウォークに対してのみ、可視であることが保証されます。

  ただし、変換テーブルへの書き込みは、プログラムの順序でその書き込みよりも前に出現する明示的なメモリアクセスからは、可視でないことが保証されます。

- 基本 ARMv7 アーキテクチャおよび ARMv7 以前のバージョンのアーキテクチャでは、変換テーブルがライトバック キャッシュ可能メモリに保持されている場合、変換テーブルへの書き込みの後、DSB 命令の前に、統一時点までキャッシュをクリーニングする必要があります。これによって、更新された変換テーブルがハードウェア変換テーブルウォークから可視になることが保証されます。

- 変換テーブルへの書き込みは、必要に応じてキャッシュからクリーニングされた後で、DSB と ISB の両方が行われた後でのみ、変換テーブルへの書き込み以後の命令フェッチにより引き起こされる変換テーブルウォークから可視であることが保証されます。

したがって、ユニプロセッサシステムの命令またはデータマッピングの変更をカバーする、変換テーブルエントリの書き込みの一般的なコードは次のとおりです。

```
STR rx, [Translation table entry]         ; write new entry to the translation table
Clean cache line [Translation table entry] : This operation is not required with the
                                           ; Multiprocessing Extensions
DSB           ; ensures visibility of the data cleaned from the D Cache
Invalidate TLB entry by MVA (and ASID if non-global) [page address]
Invalidate BTC
DSB           ; ensure completion of the Invalidate TLB operation
ISB           ; ensure table changes visible to instruction fetch
```

ASID と TTBR の変更の同期化

一般的な仮想メモリ管理の要件は、ContextID レジスタと変換テーブル ベースレジスタを同時に変更し、新しい ContextID を別の変換テーブルに関連付けることです。ただし、次のような問題から、このような変更は複雑になります。

- プリフェッチの深さが実装定義である。
- 分岐予測の使用

仮想メモリ管理操作では、ContextID レジスタと変換テーブルレジスタの変更の同期を保証する必要があります。例えば、TLB、BTC（*分岐先キャッシュ*）、および他の ASID と変換情報のキャッシュの一部またはすべてが、無効な変換によって破損する可能性があります。次のような事態を回避するため同期が必要となります。

- *古い* ASID が、*新しい* 変換テーブルからの変換テーブルウォークと関連付けられる。
- *新しい* ASID が、*古い* 変換テーブルからの変換テーブルウォークと関連付けられる。

この問題に対処するためのソリューションはいくつか存在し、最適な手法はシステムによって異なります。例 B3-2 P. B3-61、例 B3-3 P. B3-61、例 B3-4 P. B3-62 では、3 つの使用可能な手法について説明します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

Another instance of the synchronization problem occurs if a branch is encountered between changing the ASID and performing the synchronization. In this case the value in the branch predictor might be associated with the incorrect ASID. This possibility can be addressed by any of these approaches, but might be addressed by avoiding such branches.

Example B3-2 Using a reserved ASID to synchronize ASID and TTBR changes

In this approach, a particular ASID value is reserved for use by the operating system, and is used only for the synchronization of the ASID and Translation Table Base Register. This example uses the value of 0 for this purpose, but any value could be used.

This approach can be used only when the size of the mapping for any given virtual address is the same in the old and new translation tables.

The following sequence is followed, and must be executed from memory marked as being global:

```
Change ASID to 0
ISB
Change Translation Table Base Register
ISB
Change ASID to new value
```

This approach ensures that any non-global pages prefetched at a time when it is uncertain whether the old or new translation tables are being accessed are associated with the unused ASID value of 0. Since the ASID value of 0 is not used for any normal operations these entries cannot cause corruption of execution.

Example B3-3 Using translation tables that contain only global mappings when changing the ASID

A second approach involves switching the translation tables to a set of translation tables that only contain global mappings while switching the ASID.

The following sequence is followed, and must be executed from memory marked as being global:

```
Change Translation Table Base Register to the global-only mappings
ISB
Change ASID to new value
ISB
Change Translation Table Base Register to new value
```

This approach ensures that no non-global pages can be prefetched at a time when it is uncertain whether the old or new ASID value will be used.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

同期化問題のもう一つの例は、ASID の変更と同期化の実行との間で分岐が出現した場合に発生します。この場合、分岐予測器内の値が、不正な ASID に関連付けられることがあります。この問題は、これらのどの手法でも解決できますが、このような分岐を避けることによっても解決される可能性があります。

例 B3-2 予約済み ASID を使用して、ASID と TTBR の変更を同期する

この手法では、特定の ASID 値がオペレーティングシステム用に予約されます。この値は、ASID と変換テーブル ベースレジスタの同期にのみ使用されます。この例では、この目的のために 0 の値が使用されますが、任意の値を使用することができます。

この手法は、指定された仮想アドレスのマッピングサイズが、古い変換テーブルと新しい変換テーブルとで同じ場合にのみ使用できます。

シーケンスは次のとおりです。このシーケンスは、グローバルとしてマークされたメモリから実行する必要があります。

```
Change ASID to 0
ISB
Change Translation Table Base Register
ISB
Change ASID to new value
```

この手法では、古い変換テーブルと新しい変換テーブルのどちらがアクセスされているかが不明な時点でプリフェッチされた非グローバルページは、未使用の ASID 値である 0 に関連付けられることが保証されます。0 の ASID 値は通常の操作では使用されないため、これらのエントリによって実行が失敗することはありません。

例 B3-3 ASID の変更時に、グローバルマッピングのみを含む変換テーブルを使用する

2 つ目の手法では、ASID の切り替え時に、グローバルマッピングのみを含む変換テーブルのセットへ、変換テーブルを切り替えます。

シーケンスは次のとおりです。このシーケンスは、グローバルとしてマークされたメモリから実行する必要があります。

```
Change Translation Table Base Register to the global-only mappings
ISB
Change ASID to new value
ISB
Change Translation Table Base Register to new value
```

この手法では、ASID の古い値と新しい値のどちらが使用されるかが不明な時点では、非グローバルページがプリフェッチされないことが保証されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Example B3-4 Disabling non-global mappings when changing the ASID

In systems where the only non-global mappings are held in TTBR0, you can use the TTBCR.PD0 field to disable use of the TTBR0 register during the change of ASID. This means you do not require a set of global-only mappings.

The following sequence is followed, and must be executed from a memory region with a translation that is accessed from the base address in the TTBR1 register, and is marked as global:

```
Set TTBCR.PD0 = 1
ISB
Change ASID to new value
Change Translation Table Base Register to new value
ISB
Set TTBCR.PD0 = 0
```

This approach ensures that no non-global pages can be prefetched at a time when it is uncertain whether the old or new ASID value will be used.

Multiprocessor effects on TLB maintenance operations

The base ARMv7 architecture defines that the TLB maintenance operations apply only to the TLB directly attached to the processor on which the operation is executed.

To improve the implementation of multiprocessor systems, a set of extensions to ARMv7, called the Multiprocessing Extensions, has been introduced. These introduce some new TLB maintenance operations to apply to the TLBs of processors in the same Inner Shareable domain.

The extensions can be implemented in a uniprocessor system with no hardware support for cache coherency. In such a system, the Inner Shareable domain would be limited to being the single processor, and all instructions defined to apply to the Inner Shareable domain behave as aliases of the local operations.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

例 B3-4 ASID を変更時したときに非グローバルマッピングを無効化する

TTBR0 に非グローバルマッピングのみが保持されるシステムでは、TTBCR.PD0 フィールドを使用して、ASID の変更時に TTBR0 レジスタの使用を禁止することができます。これは、グローバルのみのマッピングのセットが必要ないことを意味します。

シーケンスは次のとおりです。このシーケンスは、TTBR1 レジスタのベースアドレスからアクセスされる変換を伴う、グローバルとしてマークされたメモリ領域から実行する必要があります。

```
Set TTBCR.PD0 = 1
ISB
Change ASID to new value
Change Translation Table Base Register to new value
ISB
Set TTBCR.PD0 = 0
```

この手法では、ASID の古い値と新しい値のどちらが使用されるかが不明な時点では、非グローバルページがプリフェッチされないことが保証されます。

TLB 保守操作に対するマルチプロセッサの影響

基本 ARMv7 アーキテクチャでは、TLB 保守操作が、その操作が実行されるプロセッサに直接接続された TLB に対してのみ適用されることが定義されています。

マルチプロセッサシステムの実装を強化するために、マルチプロセッシング拡張機能と呼ばれる ARMv7 に対する一連の拡張機能が導入されました。それによって、同じ内部共有可能ドメインのプロセッサの TLB に適用される、いくつかの新しい TLB 保守操作が導入されました。

これらの拡張機能は、キャッシュコヒーレンシのハードウェアサポートのないユニプロセッサシステムに実装することができます。このようなシステムでは、内部共有可能ドメインがシングルプロセッサに限定され、内部共有可能ドメインに適用するために定義されたすべての命令は、ローカル操作のエイリアスとして動作します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.11 Virtual Address to Physical Address translation operations

CP15 c7 includes operations for *Virtual Address* (VA) to *Physical Address* (PA) translation. For more information, see *CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130. The details of these operations depend on whether the Security Extensions are implemented.

All VA to PA translations take account of the TEX remapping when this remapping is enabled, see *The alternative descriptions of the Memory region attributes* on page B3-32.

A VA to PA translation operation might require a translation table walk, and an external abort might occur on this walk. For more information, see *Behavior of external aborts on a translation table walk caused by a VA to PA translation* on page B3-46. If an external abort occurs on this walk:

- The Physical Address Register, PAR:
  - — is not updated if the abort is synchronous
  - — is UNPREDICTABLE if the abort is asynchronous.
- if the Security Extensions are implemented, fault status and fault address register updates occur only in the security state in which the abort is handled. Fault address and fault status registers in the other security state are not changed.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.11 仮想アドレスから物理アドレスへの変換操作

CP15 c7 には、*仮想アドレス*(VA) から*物理アドレス*(PA) への変換用の操作が含まれています。詳細については、*CP15 c7、仮想アドレスから物理アドレスへの変換操作* P. B3- 130 を参照して下さい。これらの操作の詳細は、セキュリティ拡張機能が実装されているかどうかに依存します。

TEX 再マッピングが許可されている場合、すべての VA から PA への変換で考慮されます。*メモリ領域の属性の代替表記* P. B3- 32 を参照して下さい。

VA から PA への変換操作では変換テーブルウォークが必要な場合があり、このウォークによって外部アボードが発生する場合があります。詳細については、*VA から PA への変換によって引き起こされた変換テーブルウォークでの外部 アボートの動作* P. B3- 46 を参照して下さい。このウォークで外部アボートが発生した場合、次の動作が行われます。

- 物理アドレスレジスタ (PAR) は次のようになります。
  - – アボートが同期の場合は更新されません。
  - – アボートが非同期の場合は予測不能です。
- セキュリティ拡張機能が実装されている場合は、フォールトステータス レジスタとフォールトアドレス レジスタが、アボートが処理されるセキュリティ状態でのみ更新されます。他のセキュリティ状態のフォールトアドレス レジスタとフォールトステータス レジスタは更新されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12 CP15 registers for a VMSA implementation

This section gives a full description of the registers implemented in the CP15 System Control Coprocessor in an ARMv7 implementation that includes the VMSA memory system. Therefore, this is the description of the CP15 registers for an ARMv7-A implementation.

Some of the registers described in this section are also included in an ARMv7 implementation with a PMSA. The section *CP15 registers for a PMSA implementation* on page B4-22 also includes descriptions of these registers.

See *Coprocessors and system control* on page B1-62 for general information about the System Control Coprocessor, CP15 and the register access instructions `MRC` and `MCR`.

Information in this section is organized as follows:

- general information is given in:
  - *Organization of the CP15 registers in a VMSA implementation*
  - *General behavior of CP15 registers* on page B3-68
  - *Effect of the Security Extensions on the CP15 registers* on page B3-71
  - *Changes to CP15 registers and the memory order model* on page B3-77
  - *Meaning of fixed bit values in register diagrams* on page B3-78.
- this is followed by, for each of the primary CP15 registers c0 to c15:
  - a general description of the organization of the primary CP15 register
  - detailed descriptions of all the registers in that primary register.

Note

The detailed descriptions of the registers that implement the processor identification scheme, CPUID, are given in Chapter B5 *The CPUID Identification Scheme*, and not in this section.

Table B3-14 on page B3-66 lists all of the CP15 registers in a VMSA implementation, and is an index to the detailed description of each register.

B3.12.1 Organization of the CP15 registers in a VMSA implementation

Figure B3-10 on page B3-65 summarizes the ARMv7 CP15 registers when the VMSA is implemented. Table B3-14 on page B3-66 lists all of these registers.

Note

ARMv7 introduces significant changes to the memory system registers, especially in relation to caches. For details of:

- the CP15 register implementation in VMSAv6, see *Organization of CP15 registers for an ARMv6 VMSA implementation* on page AppxG-29
- how the ARMv7 registers must be used to discover what caches can be accessed by the processor, see *Identifying the cache resources in ARMv7* on page B2-4.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.12 VMSA 実装の CP15 レジスタ

このセクションでは、VMSA メモリシステムを含む ARMv7 実装の、CP15 システム制御コプロセッサに実装されているレジスタについて詳しく説明します。つまり、ARMv7-A 実装の CP15 レジスタについての説明です。

このセクションで説明するレジスタの一部は、PMSA を含む ARMv7 実装にも存在します。これらのレジスタについては、*PMSA 実装の CP15 レジスタ* P. B4-22 でも説明します。

システム制御コプロセッサ (CP15) と、レジスタアクセス命令 `MRC` および `MCR` の一般的な情報については、*コプロセッサとシステム制御* P. B1-62 を参照して下さい。

このセクションの情報は、次のように編成されています。

- 一般的な情報については、次のサブセクションで説明します。
  - *VMSA 実装の CP15 レジスタの編成*
  - *CP15 レジスタの一般的な動作* P. B3-68
  - *CP15 レジスタに対するセキュリティ拡張機能の影響* P. B3-71
  - *CP15 レジスタの変更とメモリオーダ モデル* P. B3-77
  - *レジスタ図の固定ビット値の意味* P. B3-78
- その後で、1 次 CP15 レジスタの c0 ～ c15 のそれぞれについて次のサブセクションで説明します。
  - 1 次 CP15 レジスタの編成についての一般的な説明
  - その 1 次レジスタに含まれるすべてのレジスタについての詳細な説明

注

プロセッサ識別方式の CPUID が実装されるレジスタの詳細については、このセクションではなく、第 B5 章「*CPUID 識別方式*」で説明します。

VMSA 実装に含まれるすべての CP15 レジスタの一覧と、各レジスタの詳細の索引を、P. B3-66 の表 B3-14 に示します。

B3.12.1 VMSA 実装の CP15 レジスタの編成

P. B3-65 の図 B3-10 は、VMSA が実装されている場合の ARMv7 CP15 レジスタをまとめたものです。これらのレジスタの一覧を、P. B3-66 の表 B3-14 に示します。

注

ARMv7 では、メモリシステム レジスタ、特にキャッシュ関連のレジスタに大幅な変更が加えられています。詳細については、次のセクションを参照して下さい。

- VMSAv6 の CP15 レジスタ実装（*ARMv6 VMSA 実装での CP15 レジスタの編成* P. 付録 G-29 を参照）
- ARMv7 レジスタを使用して、プロセッサからアクセス可能なキャッシュを特定する方法（*ARMv7 におけるキャッシュリソースの識別* P. B2-4 を参照）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure B3-10 CP15 registers in a VMSA implementation

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

図 B3- 10 VMSA 実装の CP15 レジスタ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For information about the CP15 encodings not shown in Figure B3-10 on page B3-65 see *Unpredictable and undefined behavior for CP15 accesses* on page B3-68.

Summary of CP15 register descriptions in a VMSA implementation

Table B3-14 shows the CP15 registers in a VMSA implementation. The table also includes links to the descriptions of each of the primary CP15 registers, c0 to c15.

Table B3-14 Summary of VMSA CP15 register descriptions

Register and description
CP15 c0, ID codes registers on page B3-79
c0, Main ID Register (MIDR) on page B3-81
c0, Cache Type Register (CTR) on page B3-83
c0, TCM Type Register (TCMTR) on page B3-85
c0, TLB Type Register (TLBTR) on page B3-86
c0, Multiprocessor Affinity Register (MPIDR) on page B3-87
CP15 c0, Processor Feature registers on page B5-4
c0, Debug Feature Register 0 (ID_DFR0) on page B5-6
c0, Auxiliary Feature Register 0 (ID_AFR0) on page B5-8
CP15 c0, Memory Model Feature registers on page B5-9
CP15 c0, Instruction Set Attribute registers on page B5-19
c0, Cache Size ID Registers (CCSIDR) on page B3-91
c0, Cache Level ID Register (CLIDR) on page B3-92
c0, Implementation defined Auxiliary ID Register (AIDR) on page B3-94
c0, Cache Size Selection Register (CSSELR) on page B3-95
CP15 c1, System control registers on page B3-96
c1, System Control Register (SCTLR) on page B3-96
c1, Implementation defined Auxiliary Control Register (ACTLR) on page B3-103
c1, Coprocessor Access Control Register (CPACR) on page B3-104
c1, Secure Configuration Register (SCR) on page B3-106

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

P. B3-65 の図 B3-10 に示されていない CP15 エンコードの詳細については、*CP15 アクセスの予測不能動作と未定義動作* P. B3-68 を参照して下さい。

VMSA 実装の CP15 レジスタの概要

VMSA 実装の CP15 レジスタを、表 B3-14 に示します。この表には、1 次 CP15 レジスタの c0 ～ c15 に関する説明へのリンクも含まれています。

表 B3-14 VMSA CP15 レジスタの概要

レジスタと説明
CP15 c0、ID コードレジスタ P. B3-79
c0、メイン ID レジスタ (MIDR) P. B3-81
c0、キャッシュタイプ レジスタ (CTR) P. B3-83
c0、TCM タイプレジスタ (TCMTR) P. B3-85
c0、TLB タイプレジスタ (TLBTR) P. B3-86
c0、マルチプロセッサ類似性レジスタ (MPIDR) P. B3-87
CP15 c0、プロセッサ機能レジスタ P. B5-4
c0、デバッグ機能レジスタ 0 (ID_DFR0) P. B5-6
c0、補助機能レジスタ 0 (ID_AFR0) P. B5-8
CP15 c0、メモリモデル機能レジスタ P. B5-9
CP15 c0、命令セット属性レジスタ P. B5-19
c0、キャッシュサイズ ID レジスタ (CCSIDR) P. B3-91
c0、キャッシュレベル ID レジスタ (CLIDR) P. B3-92
c0、実装定義の補助 ID レジスタ (AIDR) P. B3-94
c0、キャッシュサイズ選択レジスタ (CSSELR) P. B3-95
CP15 c1、システム制御レジスタ P. B3-96
c1、システム制御レジスタ (SCTLR) P. B3-96
c1、実装定義の補助制御レジスタ (ACTLR) P. B3-103
c1、コプロセッサアクセス制御レジスタ (CPACR) P. B3-104
c1、セキュア構成レジスタ (SCR) P. B3-106
c1、セキュアデバッグ イネーブルレジスタ (SDER) P. B3-108

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-14 Summary of VMSA CP15 register descriptions (continued)

Register and description
c1, Secure Debug Enable Register (SDER) on page B3-108
c1, Non-Secure Access Control Register (NSACR) on page B3-110
CP15 c2 and c3, Memory protection and control registers on page B3-113
c2, Translation Table Base Register 0 (TTBR0) on page B3-113
c2, Translation Table Base Register 1 (TTBR1) on page B3-116
c2, Translation Table Base Control Register (TTBCR) on page B3-117
c3, Domain Access Control Register (DACR) on page B3-119
CP15 c4, Not used on page B3-120
CP15 c5 and c6, Memory system fault registers on page B3-120
c5, Data Fault Status Register (DFSR) on page B3-121
c5, Instruction Fault Status Register (IFSR) on page B3-122
c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR) on page B3-123
c6, Data Fault Address Register (DFAR) on page B3-124
c6, Instruction Fault Address Register (IFAR) on page B3-125
CP15 c7, Cache maintenance and other functions on page B3-126
CP15 c7, Cache and branch predictor maintenance functions on page B3-126
CP15 c7, Virtual Address to Physical Address translation operations on page B3-130
CP15 c7, Data and Instruction Barrier operations on page B3-137
CP15 c7, No Operation (NOP) on page B3-138
CP15 c8, TLB maintenance operations on page B3-138
CP15 c9, Cache and TCM lockdown registers and performance monitors on page B3-141
CP15 c10, Memory remapping and TLB control registers on page B3-142
c10, Primary Region Remap Register (PRRR) on page B3-143
c10, Normal Memory Remap Register (NMRR) on page B3-146
CP15 c11, Reserved for TCM DMA registers on page B3-147

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B3- 14 VMSA CP15 レジスタの概要（続き）

レジスタと説明
c1、非セキュアアクセス制御レジスタ (NSACR) P. B3- 110
CP15 c2 および c3、メモリ保護および制御レジスタ P. B3- 113
c2、変換テーブルベースレジスタ 0 (TTBR0) P. B3- 113
c2、変換テーブルベースレジスタ 1(TTBR1) P. B3- 116
c2、変換テーブルベース制御レジスタ (TTBCR) P. B3- 117
c3、ドメインアクセス制御レジスタ (DACR) P. B3- 119
CP15 c4、未使用 P. B3- 120
CP15 c5 および c6、メモリシステム フォールトレジスタ P. B3- 120
c5、データフォールト ステータスレジスタ (DFSR) P. B3- 121
c5、命令フォールト ステータスレジスタ (IFSR) P. B3- 122
c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR) P. B3- 123
c6、データフォールト アドレスレジスタ (DFAR) P. B3- 124
c6、命令フォールト アドレスレジスタ (IFAR) P. B3- 125
CP15 c7、キャッシュ保守機能とその他の機能 P. B3- 126
CP15 c7、キャッシュおよび分岐予測器保守機能 P. B3- 126
CP15 c7、仮想アドレスから物理アドレスへの変換操作 P. B3- 130
CP15 c7、データおよび命令バリア操作 P. B3- 137
CP15 c7、無操作 (NOP) P. B3- 138
CP15 c8、TLB 保守操作 P. B3- 138
CP15 c9、キャッシュおよび TCM ロックダウンレジスタとパフォーマンスモニタ P. B3- 141
CP15 c10、メモリ再マッピングおよび TLB 制御レジスタ P. B3- 142
c10、1 次領域再マップレジスタ (PRRR) P. B3- 143
c10、ノーマルメモリ再マップレジスタ (NMRR) P. B3- 146
CP15 c11、TCM DMA 用に予約されているレジスタ P. B3- 147
CP15 c12、セキュリティ拡張機能レジスタ P. B3- 148

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-14 Summary of VMSA CP15 register descriptions (continued)

Register and description
CP15 c12, Security Extensions registers on page B3-148
c12, Vector Base Address Register (VBAR) on page B3-148
c12, Monitor Vector Base Address Register (MVBAR) on page B3-149
c12, Interrupt Status Register (ISR) on page B3-150
CP15 c13, Process, context and thread ID registers on page B3-151
c13, FCSE Process ID Register (FCSEIDR) on page B3-152
c13, Context ID Register (CONTEXTIDR) on page B3-153
CP15 c13 Software Thread ID registers on page B3-154
CP15 c14 is not used, see *Unallocated CP15 encodings* on page B3-69
CP15 c15, Implementation defined registers on page B3-155

B3.12.2 General behavior of CP15 registers

The following sections give information about the general behavior of CP15 registers:

- *Read-only bits in read/write registers*
- *Unpredictable and undefined behavior for CP15 accesses*
- *Reset behavior of CP15 registers* on page B3-70

See also *Meaning of fixed bit values in register diagrams* on page B3-78.

Read-only bits in read/write registers

Some read/write registers include bits that are read-only. These bits ignore writes.

An example of this is the SCTLR.NMFI bit, bit [27], see *c1, System Control Register (SCTLR)* on page B3-96.

UNPREDICTABLE and UNDEFINED behavior for CP15 accesses

In ARMv7 the following operations are UNDEFINED:

- all CDP, MCRR, MRRC, LDC and STC operations to CP15
- all CDP2, MCR2, MRC2, MCRR2, MRRC2, LDC2 and STC2 operations to CP15.

Unless otherwise indicated in the individual register descriptions:

- reserved fields in registers are UNK/SBZP
- reserved values of fields can have UNPREDICTABLE effects.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B3-14 VMSA CP15 レジスタの概要（続き）

レジスタと説明
c12、ベクタ ベースアドレスレジスタ (VBAR) P. B3-148
c12、モニタベクタ ベースアドレス レジスタ (MVBAR) P. B3-149
c12、割り込みステータスレジスタ (ISR) P. B3-150
CP15 c13, Process, context and thread ID registers P. B3-151
c13、FCSE プロセス ID レジスタ (FCSEIDR) P. B3-152
c13、コンテキスト ID レジスタ (CONTEXTIDR) P. B3-153
CP15 c13、ソフトウェアスレッド ID レジスタ P. B3-154
CP15 c14、未使用（*未割り当ての CP15 エンコード* P. B3-69 を参照）
CP15 c15、実装定義のレジスタ P. B3-155

B3.12.2 CP15 レジスタの一般的な動作

次のセクションでは、CP15 レジスタの一般的な動作について説明します。

- *読み出し / 書き込みレジスタの読み出し専用ビット*
- *CP15 アクセスの予測不能動作と未定義動作*
- *CP15 レジスタのリセット時の動作* P. B3-70

レジスタ図の固定ビット値の意味 P. B3-78 も参照して下さい。

読み出し / 書き込みレジスタの読み出し専用ビット

一部の読み出し / 書き込みレジスタには、読み出し専用のビットがあります。これらのビットへの書き込みは無視されます。

一例は、SCTLR.NMFI ビット（ビット [27]）です。*c1、システム制御レジスタ (SCTLR)* P. B3-96 を参照して下さい。

CP15 アクセスの予測不能動作と未定義動作

ARMv7 では、次の操作が未定義です。

- CP15 に対するすべての CDP、MCRR、MRRC、LDC、STC 操作
- CP15 に対するすべての CDP2、MCR2、MRC2、MCRR2、MRRC2、LDC2、STC2 操作

個別のレジスタの説明の中で特に記載がない場合、次の規則が適用されます。

- レジスタの予約済みフィールドは UNK/SBZP です。
- フィールドの予約済みの値は、予測不能な影響を及ぼすことがあります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The following subsections give more information about UNPREDICTABLE and UNDEFINED behavior for CP15:

- *Unallocated CP15 encodings*
- *Rules for MCR and MRC accesses to CP15 registers*
- *Effects of the Security Extensions* on page B3-70.

Unallocated CP15 encodings

When MCR and MRC instructions perform CP15 operations, the CRn value for the instruction is the major register specifier for the CP15 space. Accesses to unallocated major registers are UNDEFINED. For the ARMv7-A Architecture, this means that:

- for an implementation that includes the Security Extensions, accesses with <CRn> = {c4, c14} are UNDEFINED
- for an implementation that does not include the Security Extensions, accesses with <CRn> = {c4, c12, c14} are UNDEFINED

In an allocated CP15 major register specifier, MCR and MRC accesses to all unallocated encodings are UNPREDICTABLE for privileged accesses. For the ARMv7-A architecture this means that:

- if the Security Extensions are implemented, any privileged MCR or MRC access with <CRn> != {c4, c14} and a combination of <opc1>, <CRm> and <opc2> values not shown in Figure B3-10 on page B3-65 is UNPREDICTABLE.
- if the Security Extensions are not implemented, any privileged MCR or MRC access with <CRn> != {c4, c12, c14} and a combination of <opc1>, <CRm> and <opc2> values not shown in Figure B3-10 on page B3-65 is UNPREDICTABLE.

Note

As shown in Figure B3-10 on page B3-65, accesses to unallocated principal ID registers map onto MIDR. These are accesses with <CRn> = c0, <opc1> = 0, <CRm> = c0, and <opc2> = {4, 6, 7}.

Rules for MCR and MRC accesses to CP15 registers

All MCR operations from the PC are UNPREDICTABLE for all coprocessors, including for CP15.

All MRC operations to APSR_nzcv are UNPREDICTABLE for CP15.

The following accesses are UNPREDICTABLE:

- an MCR access to an encoding for which no write behavior is defined in any circumstances
- an MRC access to an encoding for which no read behavior is defined in any circumstances.

Except for CP15 encoding that are accessible in User mode, all MCR and MRC accesses from User mode are UNDEFINED. This applies to all User mode accesses to unallocated CP15 encodings. Individual register descriptions, and the summaries of the CP15 major registers, show the CP15 encodings that are accessible in User mode.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

次のサブセクションでは、CP15 の予測不能動作と未定義動作についてさらに詳しく説明します。

- *未割り当ての CP15 エンコード*
- *CP15 レジスタへの MCR および MRC アクセスに関する規則*
- *セキュリティ拡張機能の影響* P. B3-70

未割り当ての CP15 エンコード

MCR 命令と MRC 命令で CP15 操作を実行する場合は、命令の CRn 値が CP15 空間の主要レジスタ指定子になります。未割り当ての主要レジスタへのアクセスは、未定義です。これは、ARMv7-A アーキテクチャでは次のことを意味します。

- セキュリティ拡張機能を含む実装では、<CRn> = {c4, c14} によるアクセスは未定義です。
- セキュリティ拡張機能を含まない実装では、<CRn> = {c4, c12, c14} によるアクセスは未定義です。

割り当て済みの CP15 主要レジスタ指定子では、未割り当てのエンコードに対する MCR および MRC アクセスはすべて、特権アクセスについて予測不能です。これは、ARMv7-A アーキテクチャでは次のことを意味します。

- セキュリティ拡張機能が実装されている場合、<CRn> != {c4, c14} と、P. B3-65 の図 B3-10 に示されていない <opc1>、<CRm>、<opc2> の値の組み合わせを使用した、MCR または MRC の特権アクセスは予測不能です。
- セキュリティ拡張機能が実装されていない場合、<CRn> != {c4, c12, c14} と、P. B3-65 の図 B3-10 に示されていない <opc1>、<CRm>、<opc2> の値の組み合わせを使用した、MCR または MRC の特権アクセスは予測不能です。

注

P. B3-65 の図 B3-10 に示すように、未割り当ての基本 ID レジスタへのアクセスは、MIDR にマッピングされます。これらのアクセスは、<CRn> = c0、<opc1> = 0、<CRm> = c0、<opc2> = {4, 6, 7} を使用したアクセスです。

CP15 レジスタへの MCR および MRC アクセスに関する規則

PC からのすべての MCR 操作は、CP15 を含むすべてのコプロセッサについて予測不能です。

APSR_nzcv へのすべての MRC 操作は、CP15 について予測不能です。

次のアクセスは予測不能です。

- いずれかの状況について書き込み動作が定義されていないエンコードへの MCR アクセス
- いずれかの状況について読み出し動作が定義されていないエンコードへの MRC アクセス

ユーザモードでアクセス可能な CP15 エンコードを除いて、ユーザモードからのすべての MCR および MRC アクセスは未定義です。このことは、未割り当ての CP15 エンコードへのユーザモード アクセスにも当てはまります。個々のレジスタの説明と CP15 主要レジスタの概要には、ユーザモードでアクセス可能な CP15 エンコードが示されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Some individual registers can be made inaccessible by setting configuration bits, possibly including IMPLEMENTATION DEFINED configuration bits, to disable access to the register. The effects of the architecturally-defined configuration bits are defined individually in this manual. Typically, setting a configuration bit to disable access to a register results in the register becoming UNDEFINED for MRC and MCR accesses.

Effects of the Security Extensions

In Non-secure state, any User or privileged access to a CP15 register is UNDEFINED if either:

- There are no circumstances in which all bits and fields in the register can be accessed from Non-secure privileged modes.
- Settings in the NSACR mean that there are no circumstances in which all bits and fields in the register can be accessed from Non-secure privileged modes.

Note

The ARMv7-A architecture does not define any registers of this type. However an ARMv7-A implementation might include one or more IMPLEMENTATION DEFINED registers of this type.

When Non-secure access to a field of a CP15 register is controlled by an access control bit in the NSACR, and that access control bit is set to 0, then the controlled register field is RAZ/WI when accessed from a privileged mode in Non-secure state. If the register can be accessed from User mode then the field is also RAZ/WI when accessed from User mode.

If write access to a register is disabled by the **CP15SDISABLE** signal then any MCR access to that register is UNDEFINED.

Reset behavior of CP15 registers

After a reset, only a limited subset of the processor state is guaranteed to be set to defined values. On reset, the VMSAv7 architecture requires that the following CP15 registers are set to defined values.

Note

When the Security Extensions are implemented, only the Secure copy of a banked register is reset to the defined value.

- The SCTLR, see *c1, System Control Register (SCTLR)* on page B3-96.
- The CPACR, see *c1, Coprocessor Access Control Register (CPACR)* on page B3-104.
- The SCR, when the Security Extensions are implemented, see *c1, Secure Configuration Register (SCR)* on page B3-106.
- The TTBCR, see *c2, Translation Table Base Control Register (TTBCR)* on page B3-117.
- The Secure version of the VBAR, when the Security Extensions are implemented, see *c12, Vector Base Address Register (VBAR)* on page B3-148.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

一部のレジスタは、レジスタへのアクセスを禁止にするように構成ビット（実装定義の構成ビットが含まれる場合があります）を設定することによって、アクセス不可にすることができます。アーキテクチャ定義の構成ビットの影響は、本書内で個別に定義されます。一般的に、レジスタへのアクセスを禁止するように構成ビットを設定すると、レジスタが MRC および MCR アクセスについて未定義になります。

セキュリティ拡張機能の影響

非セキュア状態では、次のいずれかの場合に、CP15 レジスタへのユーザアクセスまたは特権アクセスは未定義です。

- レジスタのすべてのビットとフィールドが非セキュア特権モードからアクセス可能な状況が存在しない。
- NSACR の設定が、レジスタのすべてのビットとフィールドが非セキュア特権モードからアクセス可能な状況が存在しないことを意味する。

注

ARMv7-A アーキテクチャではこのタイプのレジスタが定義されていません。ただし、ARMv7-A 実装に、このタイプの実装定義のレジスタが 1 つ以上含まれている場合があります。

CP15 レジスタのフィールドへの非セキュアアクセスが NSACR のアクセス制御ビットにより制御され、そのアクセス制御ビットが 0 にセットされている場合、制御対象のレジスタフィールドが非セキュア状態の特権モードからアクセスされると、そのフィールドは RAZ/WI です。レジスタがユーザモードからアクセス可能な場合、このフィールドはユーザモードからアクセスされたときも RAZ/WI です。

レジスタへの書き込みアクセスが CP15SDISABLE 信号によって不可能にされている場合は、そのレジスタへの MCR アクセスは未定義です。

CP15 レジスタのリセット時の動作

リセット後は、プロセッサ状態の制限されたサブセットのみが、定義された値に設定されることが保証されます。VMSAv7 アーキテクチャでは、リセット時に、次の CP15 レジスタを定義された値に設定する必要があります。

注

セキュリティ拡張機能が実装されている場合は、バンクレジスタのセキュアコピーのみが定義された値にリセットされます。

- SCTLR（*c1*、*システム制御レジスタ (SCTLR)* P. B3-96 を参照）
- CPACR（*c1*、*コプロセッサアクセス制御レジスタ (CPACR)* P. B3-104 を参照）
- セキュリティ拡張機能が実装されている場合、SCR（*c1*、*セキュア構成レジスタ (SCR)* P. B3-106 を参照）
- TTBCR（*c2*、*変換テーブルベース制御レジスタ (TTBCR)* P. B3-117 を参照）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- The FCSEIDR, if the Fast Context Switch Extension (FCSE) is implemented, see *c13, FCSE Process ID Register (FCSEIDR)* on page B3-152. This register is RAZ/WI when the FCSE is not implemented.

For details of the reset values of these registers see the register descriptions. If the introductory description of a register does not include its reset value then the architecture does not require that register to be reset to a defined value.

The values of all other registers at reset are architecturally UNKNOWN. An implementation can assign an IMPLEMENTATION DEFINED reset value to a register whose reset value is architecturally UNKNOWN. After a reset, software must not rely on the value of any read/write register that does not have either an architecturally-defined reset value or an IMPLEMENTATION DEFINED reset value.

B3.12.3 Effect of the Security Extensions on the CP15 registers

When the Security Extensions are implemented, they integrate with many features of the architecture. Therefore, the descriptions of the individual CP15 registers include information about how the Security Extensions affect the register. This section:

- summarizes how the Security Extensions affect the implementation of the CP15 registers
- summarizes how the Security Extensions control access to the CP15 registers
- describes a Security Extensions signal that can control access to some CP15 registers.

It contains the following subsections:

- *Banked CP15 registers* on page B3-72
- *Restricted access CP15 registers* on page B3-73
- *Configurable access CP15 registers* on page B3-74
- *Common CP15 registers* on page B3-74
- *The CP15SDISABLE input* on page B3-76
- *Access to registers in Monitor mode* on page B3-77.

Note

- This section describes the effect of the Security Extensions on all of CP15 registers that are present in an implementation that includes the Security Extensions.
- When the Security Extensions are implemented, the register classifications of Banked, Restricted access, Configurable, or Common can apply to some coprocessor registers in addition to the CP15 registers.

It is IMPLEMENTATION DEFINED whether each IMPLEMENTATION DEFINED register is Banked, Restricted access, Configurable, or Common.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- セキュリティ拡張機能が実装されている場合、VBAR のセキュアバージョン（*c12、ベクタ ベースアドレスレジスタ (VBAR)* P. B3-148 を参照）
- 高速コンテキストスイッチ拡張機能 (FCSE) が実装されている場合、FCSEIDR（*c13、FCSE プロセス ID レジスタ (FCSEIDR)* P. B3-152 を参照）。FCSE が実装されていない場合、このレジスタは RAZ/WI です。

これらのレジスタにおけるリセット時の値の詳細については、各レジスタの説明を参照して下さい。レジスタの説明にリセット時の値が含まれていない場合は、アーキテクチャ上、そのレジスタを定義された値にリセットする必要がないことを意味します。

他のすべてのレジスタでは、リセット時の値はアーキテクチャ上未知です。実装で、リセット時の値がアーキテクチャ上未知のレジスタに、実装定義のリセット時の値を割り当てることができます。リセット後は、アーキテクチャ定義または実装定義いずれのリセット時の値も持たない読み出し / 書き込みレジスタについては、ソフトウェアで値を前提としないことが必要です。

B3.12.3 CP15 レジスタに対するセキュリティ拡張機能の影響

セキュリティ拡張機能が実装されている場合は、アーキテクチャの多くの機能と統合されます。そのため、個々の CP15 レジスタの説明には、セキュリティ拡張機能がレジスタに与える影響に関する情報が含まれています。このセクションでは、次の内容について説明します。

- セキュリティ拡張機能が CP15 レジスタの実装に与える影響の概要
- セキュリティ拡張機能によって CP15 レジスタへのアクセスが制御される方法の概要
- 一部の CP15 レジスタへのアクセスを制御可能なセキュリティ拡張機能信号

このセクションは、次のサブセクションで構成されています。

- *バンク CP15 レジスタ* P. B3-72
- *限定アクセス CP15 レジスタ* P. B3-73
- *構成可能アクセスの CP15 レジスタ* P. B3-74
- *共通の CP15 レジスタ* P. B3-74
- *CP15SDISABLE 入力* P. B3-76
- *モニタモードでのレジスタへのアクセス* P. B3-77

注

- このセクションでは、セキュリティ拡張機能を含む実装に存在する、すべての CP15 レジスタに対するセキュリティ拡張機能の影響について説明します。
- セキュリティ拡張機能が実装されている場合は、CP15 レジスタに加えて、一部のコプロセッサレジスタに、バンク、限定アクセス、構成可能、または共通のレジスタ分類を適用することができます。

実装定義の各レジスタが、バンク、限定アクセス、構成可能、共通のどれであるかは、実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Banked CP15 registers

When the Security Extensions are implemented, some CP15 registers are banked. Banked CP15 registers have two copies, one Secure and one Non-secure. The SCR.NS bit selects the Secure or Non-secure register, see *c1, Secure Configuration Register (SCR)* on page B3-106. Table B3-15 shows which registers are banked, and the permitted access to each register.

Table B3-15 Banked CP15 registers

CP15 register	Banked register	Permitted accesses[a]
c0	CSSELR, Cache Size Selection Register	Read/write in privileged modes only
c1	SCTLR, System Control Register[b]	Read/write in privileged modes only
	ACTLR, Auxiliary Control Register[c]	Read/write in privileged modes only
c2	TTBR0, Translation Table Base 0	Read/write in privileged modes only
	TTBR1, Translation Table Base 1	Read/write in privileged modes only
	TTBCR, Translation Table Base Control	Read/write in privileged modes only
c3	DACR, Domain Access Control Register	Read/write in privileged modes only
c5	DFSR, Data Fault Status Register	Read/write in privileged modes only
	IFSR, Instruction Fault Status Register	Read/write in privileged modes only
	ADFSR, Auxiliary Data Fault Status Register[c]	Read/write in privileged modes only
	AIFSR, Auxiliary Instruction Fault Status Register[c]	Read/write in privileged modes only
c6	DFAR, Data Fault Address Register	Read/write in privileged modes only
	IFAR, Instruction Fault Address Register	Read/write in privileged modes only
c7	PAR, Physical Address Register (VA to PA translation)	Read/write in privileged modes only
c10	PRRR, Primary Region Remap Register	Read/write in privileged modes only
	NMRR, Normal Memory Remap Register	Read/write in privileged modes only
c12	VBAR, Vector Base Address Register	Read/write in privileged modes only

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

バンク CP15 レジスタ

セキュリティ拡張機能が実装されている場合は、一部の CP15 レジスタがバンクされます。バンク CP15 レジスタには、セキュアと非セキュアの 2 つのコピーがあります。SCR.NS ビットによって、セキュアレジスタまたは非セキュアレジスタが選択されます。*c1、セキュア構成レジスタ (SCR)* P. B3- 106 を参照して下さい。バンクされるレジスタと、各レジスタへ許可されるアクセスを、表 B3- 15 に示します。

表 B3- 15 バンク CP15 レジスタ

CP15 レジスタ	バンクレジスタ	許可されるアクセス[a]
c0	CSSELR、キャッシュサイズ選択レジスタ	特権モードでのみ読み出し / 書き込み
c1	SCTLR、システム制御レジスタ[b]	特権モードでのみ読み出し / 書き込み
	ACTLR、補助制御レジスタ[c]	特権モードでのみ読み出し / 書き込み
c2	TTBR0、変換テーブルベース 0	特権モードでのみ読み出し / 書き込み
	TTBR1、変換テーブルベース 1	特権モードでのみ読み出し / 書き込み
	TTBCR、変換テーブルベース制御	特権モードでのみ読み出し / 書き込み
c3	DACR、ドメインアクセス制御レジスタ	特権モードでのみ読み出し / 書き込み
c5	DFSR、データフォールト ステータスレジスタ	特権モードでのみ読み出し / 書き込み
	IFSR、命令フォールト ステータスレジスタ	特権モードでのみ読み出し / 書き込み
	ADFSR、補助データフォールト ステータスレジスタ[c]	特権モードでのみ読み出し / 書き込み
	AIFSR、補助命令フォールト ステータスレジスタ[c]	特権モードでのみ読み出し / 書き込み
c6	DFAR、データフォールト アドレスレジスタ	特権モードでのみ読み出し / 書き込み
	IFAR、命令フォールト アドレスレジスタ	特権モードでのみ読み出し / 書き込み
c7	PAR、物理アドレスレジスタ（VA から PA への変換）	特権モードでのみ読み出し / 書き込み
c10	PRRR、1 次領域再マップレジスタ	特権モードでのみ読み出し / 書き込み
	NMRR、ノーマルメモリ再マップレジスタ	特権モードでのみ読み出し / 書き込み
c12	VBAR、ベクタベース アドレスレジスタ	特権モードでのみ読み出し / 書き込み

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-15 Banked CP15 registers (continued)

CP15 register	Banked register	Permitted accesses[a]
c13	FCSEIDR, FCSE PID Register[d]	Read/write in privileged modes only
	CONTEXTIDR, Context ID Register	Read/write in privileged modes only
	TPIDRURW, User Read/Write Thread ID	Read/write in unprivileged and privileged modes
	TPIDRURO, User Read-only Thread ID	Read-only in User mode Read/write in privileged modes
	TPIDRPRW, Privileged Only Thread ID	Read/write in privileged modes only

a. Any attempt to execute an access that is not permitted results in an Undefined Instruction exception.
b. Some bits are common to the Secure and the Non-secure register, see *c1, System Control Register (SCTLR)* on page B3-96.
c. Register is IMPLEMENTATION DEFINED.
d. Banked only if the FCSE is implemented. The FCSE PID Register is RAZ/WI if the FCSE is not implemented.

A Banked CP15 register can contain a mixture of:

- fields that are banked
- fields that are read-only in Non-secure privileged modes but read/write in the Secure state.

The System Control Register SCTLR is an example of a register of that contains this mixture of fields.

The Secure copies of the Banked CP15 registers are sometimes referred to as the Secure Banked CP15 registers. The Non-secure copies of the Banked CP15 registers are sometimes referred to as the Non-secure Banked CP15 registers.

Restricted access CP15 registers

When the Security Extensions are implemented, some CP15 registers are present only in the Secure security state. These are called *Restricted access* registers, and their read/write access permissions are:

- Restricted access CP15 registers cannot be modified in Non-secure state.
- The NSACR can be read in Non-secure privileged modes, but not in Non-secure User mode. This enables software running in a Non-secure privileged mode to read the access permissions for CP15 registers that have configurable access.
- Apart from the NSACR, Restricted access CP15 registers cannot be read in Non-secure state.

Table B3-16 on page B3-74 shows the Restricted access CP15 registers when the Security Extensions are implemented:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B3-15 バンク CP15 レジスタ（続き）

CP15 レジスタ	バンクレジスタ	許可されるアクセス[a]
c13	FCSEIDR、FCSE PID レジスタ[d]	特権モードでのみ読み出し / 書き込み
	CONTEXTIDR、コンテキスト ID レジスタ	特権モードでのみ読み出し / 書き込み
	TPIDRURW、ユーザ読み出し / 書き込みスレッド ID	非特権モードと特権モードで読み出し / 書き込み
	TPIDRURO、ユーザ読み出し専用スレッド ID	ユーザモードで読み出し専用 特権モードで読み出し / 書き込み
	TRIDRPRW、特権専用スレッド ID	特権モードでのみ読み出し / 書き込み

a. 許可されていないアクセスの実行を試みると、未定義命令例外が発生します。
b. 一部のビットはセキュアレジスタと非セキュアレジスタで共通です（*c1、システム制御レジスタ (SCTLR)* P. B3-96 を参照）。
c. 実装定義のレジスタです。
d. FCSE が実装されている場合にのみバンクされます。FCSE が実装されていない場合、FCSE PID レジスタは RAZ/WI です。

バンク CP15 レジスタには次の組み合わせを含めることができます。

- バンクされたフィールド
- 非セキュア特権モードでは読み出し専用だが、セキュア状態では読み出し / 書き込み可能なフィールド

システム制御レジスタ (SCTLR) は、このフィールドの組み合わせを含むレジスタの例です。

バンク CP15 レジスタのセキュアコピーは、セキュアバンク CP15 レジスタと表記される場合があります。バンク CP15 レジスタの非セキュアコピーは、非セキュアバンク CP15 レジスタと表記される場合があります。

限定アクセス CP15 レジスタ

セキュリティ拡張機能が実装されている場合は、一部の CP15 レジスタがセキュアセキュリティ状態でのみ存在します。これらのレジスタは、*限定アクセス*レジスタと呼ばれ、これらの読み出し / 書き込みアクセス許可は次のとおりです。

- 限定アクセス CP15 レジスタは、非セキュア状態では変更できません。
- NSACR は、非セキュア特権モードで読み出すことができますが、非セキュアユーザモードで読み出すことはできません。これによって、非セキュア特権モードでのソフトウェア実行で、構成可能アクセスの CP15 レジスタのアクセス許可を読み出すことができます。
- NSACR 以外の限定アクセス CP15 レジスタは、非セキュア状態で読み出せません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-16 Restricted access CP15 registers

CP15 register	Secure register	Permitted accesses[a]
c1	NSACR, Non-Secure Access Control	Read/write in Secure privileged modes Read-only in Non-secure privileged modes
	SCR, Secure Configuration	Read/write in Secure privileged modes
	SDER, Secure Debug Enable	Read/write in Secure privileged modes
c12	MVBAR, Monitor Vector Base Address	Read/write in Secure privileged modes

a. Any attempt to execute an access that is not permitted results in an Undefined Instruction exception.

Configurable access CP15 registers

Access to some CP15 registers is configurable. These registers can be:

- accessible from Secure states only
- accessible from both Secure and Non-secure states.

Access is controlled by bits in the NSACR, see *c1, Non-Secure Access Control Register (NSACR)* on page B3-110.

In ARMv7-A, the only required Configurable access CP15 register is:

- CPACR, Coprocessor Access Control Register.

Common CP15 registers

Some CP15 registers and operations are common to the Secure and Non-secure security states. These are described as the *Common access* CP15 registers, or simply as the *Common* CP15 registers. These registers are:

- Read-only registers that hold configuration information.
- Register encodings used for various memory system operations, rather than to access registers.
- The Interrupt Status Register (ISR).

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

セキュリティ拡張機能が実装されている場合の限定アクセス CP15 レジスタを、P. B3-74 の表 B3-16 に示します。

表 B3-16 限定アクセス CP15 レジスタ

CP15 レジスタ	セキュアレジスタ	許可されるアクセス[a]
c1	NSACR、非セキュアアクセス制御	セキュア特権モードで読み出し / 書き込み、 非セキュア特権モードで読み出し専用
	SCR、セキュア構成	セキュア特権モードで読み出し / 書き込み
	SDER、セキュアデバッグ イネーブル	セキュア特権モードで読み出し / 書き込み
c12	MVBAR、モニタベクタ ベースアドレス	セキュア特権モードで読み出し / 書き込み

a. 許可されていないアクセスの実行を試みると、未定義命令例外が発生します。

構成可能アクセスの CP15 レジスタ

一部の CP15 レジスタへのアクセスは構成可能です。これらのレジスタへのアクセスは、次のいずれかです。

- セキュア状態からのみアクセス可能
- セキュア状態と非セキュア状態の両方からアクセス可能

アクセスは、NSACR のビットによって制御されます。*c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3-110 を参照して下さい。

ARMv7-A で必要な構成可能アクセス CP15 レジスタは、次のものです。

- CPACR、コプロセッサアクセス制御レジスタ

共通の CP15 レジスタ

一部の CP15 レジスタおよび操作は、セキュアと非セキュアのセキュリティ状態で共通です。これらのレジスタは、*共通アクセス* CP15 レジスタ、または、単に*共通* CP15 レジスタと表記されます。これらのレジスタには次の特徴があります。

- 構成情報を保持する読み出し専用レジスタ
- レジスタのエンコードが、レジスタへのアクセス以外のさまざまなメモリシステム操作に使用される。
- 割り込みステータスレジスタ (ISR)

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-17 shows the registers that are present in an ARMv7-A implementation that are not affected by the Security Extensions. When the Security Extensions are implemented these registers are sometimes described as the *common* registers.

Table B3-17 Common CP15 registers

CP15 register	Register	Permitted accesses[a]
c0	MIDR, Main ID Register	Read-only in privileged modes only
	CTR, Cache Type Register	Read-only in privileged modes only
	TCMTR, TCM Type Register[b]	Read-only in privileged modes only
	TLBTR, TLB Type Register[b]	Read-only in privileged modes only
	MPIDR, Multiprocessor Affinity Register	Read-only in privileged modes only
	ID_PFRx, Processor Feature Registers	Read-only in privileged modes only
	ID_DFR0, Debug Feature Register 0	Read-only in privileged modes only
	ID_AFR0, Auxiliary Feature Register 0	Read-only in privileged modes only
	ID_MMFRx, Memory Model Feature Registers	Read-only in privileged modes only
	ID_ISARx, Instruction Set Attribute Registers	Read-only in privileged modes only
	CCSIDR, Cache Size ID Register	Read-only in privileged modes only
	CLIDR, Cache Level ID Register	Read-only in privileged modes only
	AIDR, Auxiliary ID Register[b]	Read-only in privileged modes only
c7	NOP	Write-only in privileged modes only
	Cache maintenance operations	See *CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
	VA to PA Translation operations	See *CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130
	Data Barrier Operations	Write-only in unprivileged and privileged modes
c8	TLB maintenance operations	Write-only in privileged modes only
c9	Performance monitors	See *Access permissions* on page C9-12
c12	ISR, Interrupt Status Register	Read-only in privileged modes only

a. Any attempt to execute an access that is not permitted results in an Undefined Instruction exception.
b. Register or operation details are IMPLEMENTATION DEFINED.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARMv7-A 実装に存在する、セキュリティ拡張機能の影響を受けないレジスタを、表 B3-17 に示します。セキュリティ拡張機能が実装されている場合、これらのレジスタは*共通*レジスタと記述されることがあります。

表 B3-17 共通 CP15 レジスタ

CP15 レジスタ	レジスタ	許可されたアクセス[a]
c0	MIDR、メイン ID レジスタ	特権モードで読み出し専用
	CTR、キャッシュタイプ レジスタ	特権モードで読み出し専用
	TCMTR、TCM タイプレジスタ[b]	特権モードで読み出し専用
	TLBTR、TLB タイプレジスタ[b]	特権モードで読み出し専用
	MPIDR、マルチプロセッサ類似性レジスタ	特権モードで読み出し専用
	ID_PFRx、プロセッサ機能レジスタ	特権モードで読み出し専用
	ID_DFR0、デバッグ機能レジスタ 0	特権モードで読み出し専用
	ID_AFR0、補助機能レジスタ 0	特権モードで読み出し専用
	ID_MMFRx、メモリモデル機能レジスタ	特権モードで読み出し専用
	ID_ISARx、命令セット属性レジスタ	特権モードで読み出し専用
	CCSIDR、キャッシュサイズ ID レジスタ	特権モードで読み出し専用
	CLIDR、キャッシュレベル ID レジスタ	特権モードで読み出し専用
	AIDR、補助 ID レジスタ[b]	特権モードで読み出し専用
c7	NOP	特権モードで書き込み専用
	キャッシュ保守操作	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126 を参照
	VA から PA への変換操作	*CP15 c7、仮想アドレスから物理アドレスへの変換操作* P. B3-130 を参照
	データバリア操作	非特権モードと特権モードで書き込み専用
c8	TLB 保守操作	特権モードで書き込み専用
c9	パフォーマンスモニタ	*アクセス許可* P. C9-12 を参照
c12	ISR、割り込みステータスレジスタ	特権モードで読み出し専用

a. 許可されていないアクセスの実行を試みると、未定義命令例外が発生します。
b. レジスタまたは操作の詳細は実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Secure CP15 registers

The Secure CP15 registers comprise:

- The Secure copies of the Banked CP15 registers
- Restricted access CP15 registers
- Configurable access CP15 registers that are configured to be accessible only from Secure state.

The CP15SDISABLE input

The Security Extensions include an input signal, **CP15SDISABLE**, that disables write access to some of the Secure registers when asserted HIGH.

Note

The interaction between **CP15SDISABLE** and any IMPLEMENTATION DEFINED register is IMPLEMENTATION DEFINED.

Table B3-18 shows the registers and operations affected.

Table B3-18 Secure registers affected by CP15SDISABLE

CP15 register	Register name	Affected operation
c1	SCTLR, System Control Register	`MCR p15, 0, <Rt>, c1, c0, 0`
c2	TTBR0, Translation Table Base Register 0	`MCR p15, 0, <Rt>, c2, c0, 0`
	TTBCR, Translation Table Base Control Register	`MCR p15, 0, <Rt>, c2, c0, 2`
c3	DACR, Domain Access Control Register	`MCR p15, 0, <Rt>, c3, c0, 0`
c10	PRRR. Primary Region Remap Register	`MCR p15, 0, <Rt>, c10, c2, 0`
	NMRR, Normal Memory Remap Register	`MCR p15, 0, <Rt>, c10, c2, 1`
c12	VBAR, Vector Base Address Register	`MCR p15, 0, <Rt>, c12, c0, 0`
	MVBAR, Monitor Vector Base Address Register	`MCR p15, 0, <Rt>, c12, c0, 1`
c13	FCSEIDR, FCSE PID Register [a]	`MCR p15, 0, <Rt>, c13, c0, 0`

a. If the FCSE is implemented. The FCSE PID Register is RAZ/WI if the FCSE is not implemented.

On a reset by the external system, the **CP15SDISABLE** input signal must be taken LOW. This permits the Reset code to set up the configuration of the Security Extensions. When the input is asserted HIGH, any attempt to write to the Secure registers shown in Table B3-18 results in an Undefined Instruction exception.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

セキュア CP15 レジスタ

セキュア CP15 レジスタには、次のものが含まれます。

- バンク CP15 レジスタのセキュアコピー
- 限定アクセス CP15 レジスタ
- セキュア状態からのみアクセス可能であるように構成されている、構成可能アクセス CP15 レジスタ

CP15SDISABLE 入力

セキュリティ拡張機能には CP15SDISABLE 入力信号が含まれています。この信号が HIGH にアサートされると、一部のセキュアレジスタへの書き込みアクセスが禁止されます。

注

CP15SDISABLE と任意の実装定義のレジスタとの相互動作は、実装定義です。

影響を受けるレジスタと操作を、表 B3-18 に示します。

表 B3-18 CP15SDISABLE の影響を受けるセキュアレジスタ

CP15 レジスタ	レジスタ名	影響を受ける操作
c1	SCTLR、システム制御レジスタ	`MCR p15, 0, <Rt>, c1, c0, 0`
c2	TTBR0、変換テーブル ベースレジスタ 0	`MCR p15, 0, <Rt>, c2, c0, 0`
	TTBCR、変換テーブルベース制御レジスタ	`MCR p15, 0, <Rt>, c2, c0, 2`
c3	DACR、ドメインアクセス制御レジスタ	`MCR p15, 0, <Rt>, c3, c0, 0`
c10	PRRR、1 次領域再マップレジスタ	`MCR p15, 0, <Rt>, c10, c2, 0`
	NMRR、ノーマルメモリ再マップレジスタ	`MCR p15, 0, <Rt>, c10, c2, 1`
c12	VBAR、ベクタ ベースアドレス レジスタ	`MCR p15, 0, <Rt>, c12, c0, 0`
	MVBAR、モニタベクタ ベースアドレス レジスタ	`MCR p15, 0, <Rt>, c12, c0, 1`
c13	FCSEIDR、FCSE PID レジスタ [a]	`MCR p15, 0, <Rt>, c13, c0, 0`

a. FCSE が実装されている場合。FCSE が実装されていない場合、FCSE PID レジスタは RAZ/WI です。

外部システムによるリセット時に、CP15SDISABLE 入力信号を LOW にする必要があります。これによって、リセットコードでセキュリティ拡張機能の構成をセットアップすることができます。この入力が HIGH にアサートされているときに、表 B3-18 に示すセキュアレジスタに書き込みを試みると、未定義命令例外が発生します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The **CP15SDISABLE** input does not affect reading Secure registers, or reading or writing Non-secure registers. It is IMPLEMENTATION DEFINED how the input is changed and when changes to this input are reflected in the processor. However, changes must be reflected as quickly as possible. The change must occur before completion of a Instruction Synchronization Barrier operation, issued after the change, is visible to the processor with respect to instruction execution boundaries. Software must perform a Instruction Synchronization Barrier operation meeting the above conditions to ensure all subsequent instructions are affected by the change to **CP15SDISABLE**.

The assertion of **CP15SDISABLE** enables key Secure privileged features to be locked in a known good state, providing an additional level of overall system security. ARM expects control of this input to reside in the system, in a system block dedicated to security.

Access to registers in Monitor mode

When the processor is in Monitor mode, the processor is in Secure state regardless of the value of the SCR.NS bit. In Monitor mode, the SCR.NS bit determines whether the Secure Banked CP15 registers or Non-secure Banked CP15 registers are read or written using MRC or MCR instructions. That is:

NS = 0 Common, Restricted access, and Secure Banked registers are accessed by CP15 MRC and MCR instructions.

CP15 operations use the security state to determine all resources used, that is, all CP15 based operations are performed in Secure state.

NS = 1 Common, Restricted access and Non-secure Banked registers are accessed by CP15 MRC and MCR instructions.

CP15 operations use the security state to determine all resources used, that is, all CP15 based operations are performed in Secure state.

The security state determines whether the Secure or Non-secure Banked registers are used to determine the control state.

B3.12.4 Changes to CP15 registers and the memory order model

All changes to CP15 registers that appear in program order after any explicit memory operations are guaranteed not to affect those memory operations.

Any change to CP15 registers is guaranteed to be visible to subsequent instructions only after one of:

- the execution of an ISB instruction
- the taking of an exception
- the return from an exception.

To guarantee the visibility of changes to some CP15 registers, additional operations might be required, on a case by case basis, before the ISB instruction, exception or return from exception. These cases are identified specifically in the definition of the registers.

However, for CP15 register accesses, all MRC and MCR instructions to the same register using the same register number appear to occur in program order relative to each other without context synchronization.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP15SDISABLE 入力は、セキュアレジスタの読み出しまたは非セキュアレジスタの読み出し / 書き込みに影響を与えません。この入力の変更方法と、この入力の変更がプロセッサに反映されるタイミングは、実装定義です。ただし、変更はできるだけ迅速に反映される必要があります。この変更は、変更後に発行される命令同期バリア操作の完了が命令実行境界でプロセッサに可視になる前に、発生する必要があります。ソフトウェアは、後続のすべての命令が CP15SDISABLE の変更の影響を受けることを保証するため、上記の条件を満たす命令同期バリア操作を実行する必要があります。

CP15SDISABLE のアサートにより、主要なセキュア特権機能を既知の良好な状態でロックして、新たなレベルのシステム全体のセキュリティを提供することができます。この入力の制御は、システム内のセキュリティに特化されたシステムブロックに存在することが想定されます。

モニタモードでのレジスタへのアクセス

プロセッサがモニタモードのときは、SCR.NS ビットの値にかかわらず、プロセッサはセキュア状態にあります。モニタモードでは、SCR.NS ビットによって、MRC 命令または MCR 命令で、セキュアバンク CP15 レジスタと非セキュアバンク CP15 レジスタのどちらが読み書きされるかが決定されます。これは、次のことを意味します。

NS = 0 共通、限定アクセス、セキュアのバンクレジスタが、CP15 の MRC および MCR 命令によってアクセスされます。

CP15 操作では、セキュリティ状態を使用して、使用されるすべてのリソースが決定されます。つまり、すべての CP15 ベースの操作はセキュア状態で実行されます。

NS = 1 共通、限定アクセス、非セキュアのバンクレジスタが、CP15 の MRC および MCR 命令によってアクセスされます。

CP15 操作では、セキュリティ状態を使用して、使用されるすべてのリソースが決定されます。つまり、すべての CP15 ベースの操作はセキュア状態で実行されます。

セキュリティ状態によって、セキュアバンク レジスタと非セキュアバンク レジスタのどちらが制御状態の決定に使用されるかが決定されます。

B3.12.4 CP15 レジスタの変更とメモリオーダ モデル

プログラムの順序で明示的なメモリ操作の後に出現する CP15 レジスタの変更は、それらのメモリ操作に影響を与えないことが保証されます。

CP15 レジスタの変更は、次のいずれかの後でのみ、以後の命令から可視であることが保証されます。

- ISB 命令の実行
- 例外の取得
- 例外からの復帰

一部の CP15 レジスタの変更の可視性を保証するには、ISB 命令、例外、または例外からの復帰の前に、条件に応じて追加の操作が必要な場合があります。このような条件は、レジスタの定義で明確に特定されます。

ただし、CP15 レジスタへのアクセスでは、同じレジスタ番号を使用した同じレジスタへの MRC 命令と MCR 命令のすべてが、相対的にプログラムでの順序で、コンテキスト同期なしに発生するように見えます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Where a change to the CP15 registers that is not yet guaranteed to be visible has an effect on exception processing, the following rule applies:

- When it is determined that an exception must be taken, any change of state held in CP15 registers involved in the triggering of the exception and that affects the processing of the exception is guaranteed to take effect before the exception is taken.

Therefore, in the following example, where initially A=1 and V=0, the LDR might or might not take a Data Abort exception due to the unaligned access, but if an exception occurs the vector used is affected by the V bit:

```
MCR p15, R0, c1, c0, 0    ; clears the A bit and sets the V bit
LDR R2, [R3]              ; unaligned load.
```

B3.12.5 Meaning of fixed bit values in register diagrams

In register diagrams, fixed bits are indicated by one of following:

0 In any implementation:
- the bit must read as 0
- writes to the bit must be ignored.

Software:
- can rely on the bit reading as 0
- must use an SBZP policy to write to the bit.

(0) In any implementation:
- the bit must read as 0
- writes to the bit must be ignored.

Software:
- must not rely on the bit reading as 0
- must use an SBZP policy to write to the bit.

1 In any implementation:
- the bit must read as 1
- writes to the bit must be ignored.

Software:
- can rely on the bit reading as 1
- must use an SBOP policy to write to the bit.

(1) In any implementation:
- the bit must read as 1
- writes to the bit must be ignored.

Software:
- must not rely on the bit reading as 1
- must use an SBOP policy to write to the bit.

Fields that are more than 1 bit wide are sometimes described as UNK/SBZP, instead of having each bit marked as (0).

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

可視であることが保証されていない CP15 レジスタの変更が例外処理に影響を与える場合、次の規則が適用されます。

- 例外を取得する必要があることが判明した場合、CP15 レジスタに保持されている状態の変更のうち、例外のトリガに関係したものと、例外の処理に影響を与えるものは、その例外が取得される前に反映されることが保証されます。

したがって、次の例では、初期状態で A = 1 および V = 0 の場合は、LDR によってアンアラインドアクセスが原因のデータアボート例外が取得される場合とされない場合がありますが、例外が発生した場合は、使用されるベクタが V ビットの影響を受けます。

```
MCR p15, R0, c1, c0, 0    ; clears the A bit and sets the V bit
LDR R2, [R3]              ; unaligned load
```

B3.12.5 レジスタ図の固定ビット値の意味

レジスタ図では、固定ビットは次のいずれかによって示されます。

0 任意の実装において
- ビットは 0 として読み出される必要がある。
- ビットへの書き込みは無視される必要がある。

ソフトウェアでは
- ビットが 0 として読み出されることを前提とできる。
- SBZP ポリシーを使用してビットに書き込む必要がある。

(0) 任意の実装において
- ビットは 0 として読み出される必要がある。
- ビットへの書き込みは無視される必要がある。

ソフトウェアでは
- ビットが 0 として読み出されることを前提とできない。
- SBZP ポリシーを使用してビットに書き込む必要がある。

1 任意の実装において
- ビットは 1 として読み出される必要がある。
- ビットへの書き込みは無視される必要がある。

ソフトウェアでは
- ビットが 1 として読み出されることを前提とできる。
- SBOP ポリシーを使用してビットに書き込む必要がある。

(1) 任意の実装において
- ビットは 1 として読み出される必要がある。
- ビットへの書き込みは無視される必要がある。

ソフトウェアでは
- ビットが 1 として読み出されることを前提とできない。
- SBOP ポリシーを使用してビットに書き込む必要がある。

1 ビット幅を超えるフィールドは、各ビットを (0) としてマークする代わりに、UNK/SBZP として表記される場合があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.6 CP15 c0, ID codes registers

The CP15 c0 registers are used for processor and feature identification. Figure B3-11 shows the CP15 c0 registers.

Figure B3-11 CP15 c0 registers in a VMSA implementation

CP15 c0 register encodings not shown in Figure B3-11 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B3-69.

Note

Chapter B5 *The CPUID Identification Scheme* describes the CPUID registers shown in Figure B3-11.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.6 CP15 c0、ID コードレジスタ

CP15 c0 レジスタは、プロセッサと機能の識別に使用されます。CP15 c0 レジスタを、図 B3-11 に示します。

図 B3-11 VMSA 実装の CP15 c0 レジスタ

図 B3-11 に示されていない CP15 c0 レジスタのエンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B3-69 を参照して下さい。

注

図 B3-11 に示されている CPUID レジスタについては、第 B5 章 *「CPUID 識別方式」* で説明します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-19 lists the CP15 c0 registers and shows where each register is described in full. The table does not include the reserved and aliased registers that are shown in Figure B3-11 on page B3-79.

Table B3-19 Index to CP15 c0 register descriptions

opc1	CRm	opc2	Register and description
0	c0	0	*c0, Main ID Register (MIDR)* on page B3-81
		1	*c0, Cache Type Register (CTR)* on page B3-83
		2	*c0, TCM Type Register (TCMTR)* on page B3-85
		3	*c0, TLB Type Register (TLBTR)* on page B3-86
		5	*c0, Multiprocessor Affinity Register (MPIDR)* on page B3-87
		4, 6, 7	*c0, Main ID Register (MIDR)* on page B3-81
	c1	0, 1	*CP15 c0, Processor Feature registers* on page B5-4
		2	*c0, Debug Feature Register 0 (ID_DFR0)* on page B5-6
		3	*c0, Auxiliary Feature Register 0 (ID_AFR0)* on page B5-8
		4-7	*CP15 c0, Memory Model Feature registers* on page B5-9
	c2	0-5	*CP15 c0, Instruction Set Attribute registers* on page B5-19
1	c0	0	*c0, Cache Size ID Registers (CCSIDR)* on page B3-91
		1	*c0, Cache Level ID Register (CLIDR)* on page B3-92
		7	*c0, Implementation defined Auxiliary ID Register (AIDR)* on page B3-94
2	c0	0	*c0, Cache Size Selection Register (CSSELR)* on page B3-95

Note

The CPUID scheme described in Chapter B5 *The CPUID Identification Scheme* includes information about the implementation of the optional VFP and Advanced SIMD architecture extensions. See *Advanced SIMD and VFP extensions* on page A2-20 for a summary of the implementation options for these features.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

CP15 c0 レジスタと、各レジスタについて完全な説明のあるセクションを、表 B3-19 に示します。この表には、P. B3-79 の図 B3-11 に示す予約済みのレジスタとエイリアスレジスタは含まれていません。

表 B3-19 CP15 c0 レジスタの説明の索引

opc1	CRm	opc2	レジスタと説明
0	c0	0	*c0、メイン ID レジスタ (MIDR)* P. B3-81
		1	*c0、キャッシュタイプ レジスタ (CTR)* P. B3-83
		2	*c0、TCM タイプレジスタ (TCMTR)* P. B3-85
		3	*c0、TLB タイプレジスタ (TLBTR)* P. B3-86
		5	*c0、マルチプロセッサ類似性レジスタ (MPIDR)* P. B3-87
		4, 6, 7	*c0、メイン ID レジスタ (MIDR)* P. B3-81
	c1	0, 1	*CP15 c0、プロセッサ機能レジスタ* P. B5-4
		2	*c0、デバッグ機能レジスタ 0 (ID_DFR0)* P. B5-6
		3	*c0、補助機能レジスタ 0 (ID_AFR0)* P. B5-8
		4-7	*CP15 c0、メモリモデル機能レジスタ* P. B5-9
	c2	0-5	*CP15 c0、命令セット属性レジスタ* P. B5-19
1	c0	0	*c0、キャッシュサイズ ID レジスタ (CCSIDR)* P. B3-91
		1	*c0、キャッシュレベル ID レジスタ (CLIDR)* P. B3-92
		7	*c0、実装定義の補助 ID レジスタ (AIDR)* P. B3-94
2	c0	0	*c0、キャッシュサイズ選択レジスタ (CSSELR)* P. B3-95

注

第 B5 章 *「CPUID 識別方式」* で説明されている CPUID 方式には、オプションの VFP およびアドバンスト SIMD アーキテクチャ拡張機能の実装に関する情報が含まれています。これらの機能の実装オプションの概要については、*アドバンスト SIMD および VFP 拡張機能* P. A2-20 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.7 c0, Main ID Register (MIDR)

The Main ID Register, MIDR, provides identification information for the processor, including an implementer code for the device and a device ID number.

The MIDR is:

- a 32-bit read-only register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Common register.

Some fields of the MIDR are IMPLEMENTATION DEFINED. For details of the values of these fields for a particular ARMv7 implementation, and any implementation-specific significance of these values, see the product documentation.

The format of the MIDR is:

31 – 24	23 – 20	19 – 16	15 – 4	3 – 0
Implementer	Variant	Architecture	Primary part number	Revision

Implementer, bits [31:24]

The Implementer code. Table B3-20 shows the permitted values for this field:

Table B3-20 Implementer codes

Bits [31:24]	ASCII character	Implementer
0x41	A	ARM Limited
0x44	D	Digital Equipment Corporation
0x4D	M	Motorola, Freescale Semiconductor Inc.
0x51	Q	QUALCOMM Inc.
0x56	V	Marvell Semiconductor Inc.
0x69	i	Intel Corporation

All other values are reserved by ARM and must not be used.

Variant, bits [23:20]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.7 c0、メイン ID レジスタ (MIDR)

メイン ID レジスタ (MIDR) では、デバイスの実装者コードやデバイス ID 番号を含む、プロセッサの識別情報が提供されます。

MIDR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

MIDR の一部のフィールドは、実装定義です。特定の ARMv7 実装におけるこれらのフィールドの値と、これらの値の実装固有の意味については、製品説明書を参照して下さい。

MIDR のフォーマットは次のとおりです。

31 – 24	23 – 20	19 – 16	15 – 4	3 – 0
実装者	バリアント	アーキテクチャ	主要部品番号	リビジョン

実装者、ビット [31:24]

実装者コード。このフィールドに許可される値を、表 B3-20 に示します。

表 B3-20 実装者コード

ビット [31:24]	ASCII 文字	実装者
0x41	A	ARM Limited
0x44	D	Digital Equipment Corporation
0x4D	M	Motorola、Freescale Semiconductor Inc.
0x51	Q	QUALCOMM Inc.
0x56	V	Marvell Semiconductor Inc.
0x69	i	Intel Corporation

他のすべての値は ARM によって予約されており、使用できません。

バリアント、ビット [23:20]

実装定義のバリアント番号。一般に、このフィールドは、複数の製品バリアントまたは製品のメジャーリビジョンを区別するために使用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Architecture, bits [19:16]

Table B3-21 shows the permitted values for this field:

Table B3-21 Architecture codes

Bits [19:16]	Architecture
0x1	ARMv4
0x2	ARMv4T
0x3	ARMv5 (obsolete)
0x4	ARMv5T
0x5	ARMv5TE
0x6	ARMv5TEJ
0x7	ARMv6
0xF	Defined by CPUID scheme

All other values are reserved by ARM and must not be used.

Primary part number, bits [15:4]

An IMPLEMENTATION DEFINED primary part number for the device.

Note

On processors implemented by ARM, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently, see *c0, Main ID Register (MIDR)* on page AppxH-34. Processors implemented by ARM have an Implementer code of 0x41.

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the device.

ARMv7 requires all implementations to use the CPUID scheme, described in Chapter B5 *The CPUID Identification Scheme*, and an implementation is described by the MIDR with the CPUID registers.

Note

For an ARMv7 implementation by ARM, the MIDR is interpreted as:

Bits [31:24] Implementer code, must be 0x41.
Bits [23:20] Major revision number, rX.
Bits [19:16] Architecture code, must be 0xF.
Bits [15:4] ARM part number.
Bits [3:0] Minor revision number, pY.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アーキテクチャ、ビット [19:16]

このフィールドに許可される値を、表 B3-21 に示します。

表 B3-21 アーキテクチャコード

ビット [19:16]	アーキテクチャ
0x1	ARMv4
0x2	ARMv4T
0x3	ARMv5（廃止）
0x4	ARMv5T
0x5	ARMv5TE
0x6	ARMv5TEJ
0x7	ARMv6
0xF	CPUID 方式で定義

他のすべての値は ARM によって予約されており、使用できません。

主要部品番号、ビット [15:4]

デバイスの、実装定義の主要部品番号。

注

ARM によって実装されたプロセッサでは、主要部品番号の上位 4 ビットが 0x0 または 0x7 の場合は、バリアントとアーキテクチャについて異なるエンコードが使用されます。*c0、メイン ID レジスタ (MIDR)* P. 付録 H-34 を参照して下さい。ARM によって実装されたプロセッサの実装者コードは 0x41 です。

リビジョン、ビット [3:0]

デバイスの、実装定義のリビジョン番号。

ARMv7 では、すべての実装において、第 B5 章 *「CPUID 識別方式」* で説明されている CPUID 方式を使用する必要があります。実装は、MIDR レジスタと CPUID レジスタにより記述されます。

注

ARM による ARMv7 実装の場合は、MIDR が次のように解釈されます。

ビット [31:24] 実装者コード、0x41 にする必要があります。
ビット [23:20] メジャーリビジョン番号、rX
ビット [19:16] アーキテクチャコード、0xF にする必要があります。
ビット [15:4] ARM 部品番号
ビット [3:0] マイナーリビジョン番号、pY

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the MIDR

To access the MIDR you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c0,c0,0    ; Read CP15 Main ID Register
```

B3.12.8 c0, Cache Type Register (CTR)

The Cache Type Register, CTR, provides information about the architecture of the caches.

The CTR is:

- a 32-bit read-only register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Common register.

The format of the CTR is changed in ARMv7. The new format of the register is indicated by Bit [31:29] being set to 0b100. For details of the format of the Cache Type Register in versions of the ARM architecture before ARMv7 see *c0, Cache Type Register (CTR)* on page AppxH-35.

In ARMv7, the format of the CTR is:

31	29	28	27	24	23	20	19	16	15	14	13							4	3	0
1	0	0	0	CWG		ERG		DminLine	L1Ip	0	0	0	0	0	0	0	0	0	IminLine	

Bits [31:29] Set to 0b100 for the ARMv7 register format. Set to 0b000 for the format used in ARMv6 and earlier.

Bit [28] RAZ.

CWG, bits [27:24]

Cache Writeback Granule. Log_2 of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

A value of 0b0000 indicates that the CTR does not provide Cache Writeback Granule information and either:

- the architectural maximum of 512 words (2Kbytes) must be assumed
- the Cache Writeback Granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

Values greater than 0b1001 are reserved.

ERG, bits [27:24]

Exclusives Reservation Granule. Log_2 of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions. For more information, see *Tagging and the size of the tagged memory block* on page A3-20.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

MIDR へのアクセス

MIDR にアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,0,<Rt>,c0,c0,0    ; Read CP15 Main ID Register
```

B3.12.8 c0、キャッシュタイプ レジスタ (CTR)

キャッシュタイプ レジスタ (CTR) は、キャッシュのアーキテクチャに関する情報を提供します。

CTR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

CTR のフォーマットは ARMv7 で変更されました。このレジスタの新しいフォーマットは、ビット [31:29] が 0b100 にセットされていることで示されます。ARMv7 以前のバージョンの ARM アーキテクチャにおけるキャッシュタイプ レジスタのフォーマットの詳細については、*c0、キャッシュタイプ レジスタ (CTR)* P. 付録 H- 35 を参照して下さい。

ARMv7 における CTR のフォーマットは次のとおりです。

31	29	28	27	24	23	20	19	16	15	14	13							4	3	0
1	0	0	0	CWG		ERG		DminLine	L1Ip	0	0	0	0	0	0	0	0	0	IminLine	

ビット [31:29] ARMv7 レジスタフォーマットの場合は、0b100 にセットします。ARMv6 およびそれ以前で使用されるフォーマットの場合は、0b000 にセットします。

ビット [28] RAZ

CWG、ビット [27:24]

キャッシュライトバックの粒度。キャッシュエントリに含まれるメモリ位置が変更された場合、そのキャッシュエントリの追い出しにより上書きされる可能性があるメモリの最大サイズを、ワード単位で示した数値の Log_2 です。

0b0000 の値は、CTR でキャッシュライトバックの粒度情報が提供されず、次のいずれかであることを意味します。

- アーキテクチャ上の上限である、512 ワード (2KB) を前提とする必要がある。
- キャッシュライトバックの粒度は、キャッシュサイズ ID レジスタにエンコードされている最大キャッシュライン サイズから特定できる。

0b1001 より大きい値は予約されています。

ERG、ビット [23:20]

排他予約の粒度。排他ロード命令と排他ストア命令用に実装されている予約粒度の最大サイズを、ワード単位で示した数値の Log_2 です。詳細については、*タグ付けとタグ付けされたメモリブロックのサイズ* P. A3- 20 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A value of 0b0000 indicates that the CTR does not provide Exclusives Reservation Granule information and the architectural maximum of 512 words (2Kbytes) must be assumed.

Values greater than 0b1001 are reserved.

DminLine, bits [19:16]

Log_2 of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the processor.

L1Ip, bits [15:14]

Level 1 instruction cache policy. Indicates the indexing and tagging policy for the L1 instruction cache. Table B3-22 shows the possible values for this field.

Table B3-22 Level 1 instruction cache policy field values

L1Ip bits	L1 instruction cache indexing and tagging policy
00	Reserved
01	*ASID-tagged Virtual Index, Virtual Tag* (AIVIVT)
10	*Virtual Index, Physical Tag* (VIPT)
11	*Physical Index, Physical Tag* (PIPT)

Bits [13:4] RAZ.

IminLine, bits [3:0]

Log_2 of the number of words in the smallest cache line of all the instruction caches that are controlled by the processor.

Accessing the CTR

To access the CTR you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 1. For example

```
MRC p15,0,<Rt>,c0,c0,1    ; Read CP15 Cache Type Register
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

0b0000 の値は、CTR で排他予約の粒度情報が提供されず、アーキテクチャ上の上限である 512 ワード (2KB) を前提とする必要があることを意味します。

0b1001 より大きい値は予約されています。

DminLine、ビット [19:16]

プロセッサで制御されるすべてのデータキャッシュと統一キャッシュの、最小キャッシュラインのワード数の Log_2 です。

L1Ip、ビット [15:14]

レベル 1 命令キャッシュポリシー。L1 命令キャッシュの索引付けおよびタグ付けポリシーを示します。このフィールドに許可される値を、表 B3-22 に示します。

表 B3-22 レベル 1 命令キャッシュポリシー フィールドの値

L1Ip ビット	L1 命令キャッシュの索引付けおよびタグ付けポリシー
00	予約
01	*ASID タグ付き仮想インデクス仮想タグ* (AIVIVT)
10	*仮想インデクス物理タグ* (VIPT)
11	*物理インデクス物理タグ* (PIPT)

ビット [13:4] RAZ

IminLine、ビット [3:0]

プロセッサで制御されるすべての命令キャッシュの、最小キャッシュラインのワード数の Log_2 です。

CTR へのアクセス

CTR にアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 1 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,0,<Rt>,c0,c0,1    ; Read CP15 Cache Type Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.9 c0, TCM Type Register (TCMTR)

The TCM Type Register, TCMTR, provides information about the implementation of the TCM.

The TCMTR is:
- a 32-bit read-only register
- accessible only in privileged modes.
- when the Security Extensions are implemented, a Common register.

From ARMv7:
- TCMTR must be implemented
- when the ARMv7 format is used, the meaning of register bits [28:0] is IMPLEMENTATION DEFINED
- the ARMv6 format of the TCM Type Register remains a valid usage model
- if no TCMs are implemented the ARMv6 format must be used to indicate zero-sized TCMs.

The ARMv7 format of the TCMTR is:

31 29	28 0
1 0 0	IMPLEMENTATION DEFINED

Bits [31:29] Set to 0b100 for the ARMv7 register format.

Note

This field is set to 0b000 for the format used in ARMv6 and earlier.

Bits [28:0] IMPLEMENTATION DEFINED in the ARMv7 register format.

If no TCMs are implemented, the TCMTR must be implemented with this ARMv6 format:

31 29	28 19	18 16	15 3	2 0
0 0 0	UNKNOWN	0 0 0	UNKNOWN	0 0 0

For details of the ARMv6 optional implementation of the TCM Type Register see *c0, TCM Type Register (TCMTR)* on page AppxG-33.

Accessing the TCMTR

To access the TCMTR you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 2. For example:

```
MRC p15,0,<Rt>,c0,c0,2    ; Read CP15 TCM Type Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.9 c0、TCM タイプレジスタ (TCMTR)

TCM タイプレジスタ (TCMTR) は、TCM の実装に関する情報を提供します。

TCMTR の特徴は次のとおりです。
- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

ARMv7 およびそれ以降では、次の規則が適用されます。
- TCMTR を実装する必要があります。
- ARMv7 フォーマットが使用されている場合、レジスタのビット [28:0] の意味は実装定義です。
- TCM タイプレジスタの ARMv6 フォーマットは、有効な使用モデルとして残されています。
- TCM が実装されていない場合は、ARMv6 フォーマットを使用してサイズ 0 の TCM を示す必要があります。

TCMTR の ARMv7 フォーマットは次のとおりです。

31 29	28 0
1 0 0	実装定義

ビット [31:29] ARMv7 レジスタフォーマットの場合は、0b100 にセットします。

注

このフィールドは、ARMv6 およびそれ以前で使用されるフォーマットの場合は 0b000 にセットされます。

ビット [28:0] ARMv7 レジスタフォーマットでは実装定義です。

TCM が実装されていない場合は、この ARMv6 フォーマットを使用して TCMTR を実装する必要があります。

31 29	28 19	18 16	15 3	2 0
0 0 0	未知	0 0 0	未知	0 0 0

TCM タイプレジスタの ARMv6 オプション実装の詳細については、*c0、TCM タイプレジスタ (TCMTR)* P. 付録 G-33 を参照して下さい。

TCMTR へのアクセス

TCMTR にアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 2 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,0,<Rt>,c0,c0,2    ; Read CP15 TCM Type Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.10 c0, TLB Type Register (TLBTR)

The TLB Type Register, TLBTR, provides information about the TLB implementation. The register must define whether the implementation provides separate instruction and data TLBs, or a unified TLB. Normally, the IMPLEMENTATION DEFINED information in this register includes the number of lockable entries in the TLB.

The TLBTR is:

- a 32-bit read-only register
- accessible only in privileged modes
- implemented only when the VMSA is implemented
- when the Security Extensions are implemented, a Common register.

The format of the TLBTR is:

31 … 1	0
IMPLEMENTATION DEFINED	nU

Bits [31:1] IMPLEMENTATION DEFINED.

nU, bit [0] Not Unified TLB. Indicates whether the implementation has a unified TLB:

nU == 0 Unified TLB.

nU == 1 Separate Instruction and Data TLBs.

Note

From ARMv7, the TLB lockdown mechanism is IMPLEMENTATION DEFINED, and therefore the details of bits [31:1] of the TLB Type Register are IMPLEMENTATION DEFINED.

Accessing the TLBTR

To access the TLBTR you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 3. For example:

```
MRC p15,0,<Rt>,c0,c0,3    ; Read CP15 TLB Type Register
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.12.10 c0、TLB タイプレジスタ (TLBTR)

TLB タイプレジスタ (TLBTR) は、TLB 実装に関する情報を提供します。このレジスタを使用して、実装で独立した命令 TLB とデータ TLB を提供するのか、統一 TLB を提供するのかを定義する必要があります。通常は、このレジスタ内の実装定義情報に、TLB のロック可能エントリの数が含まれています。

TLBTR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- VMSA が実装されている場合にのみ実装されます。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

TLBTR のフォーマットは次のとおりです。

31 … 1	0
実装定義	nU

ビット [31:1] 実装定義

nU、ビット [0]

統一 TLB の非実装。実装に統一 TLB が含まれているかどうかを示します。

nU = 0 統一 TLB

nU = 1 独立した命令 TLB とデータ TLB

注

ARMv7 およびそれ以降では、TLB ロックダウン機構は実装定義なので、TLB タイプレジスタのビット [31:1] の詳細は実装定義です。

TLBTR へのアクセス

TLBTR にアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 3 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,0,<Rt>,c0,c0,3    ; Read CP15 TLB Type Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.11 c0, Multiprocessor Affinity Register (MPIDR)

The Multiprocessor Affinity Register, MPIDR, provides an additional processor identification mechanism for scheduling purposes in a multiprocessor system. In a uniprocessor system ARM recommends that this register returns a value of 0.

The MPIDR is:

- a 32-bit read-only register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Common register
- introduced in ARMv7.

In the ARMv7 base architecture the format of the MPIDR is:

When the Multiprocessing Extensions are implemented the format of the MPIDR is:

Note

In the MIDR bit definitions, a *processor in the system* can be a physical processor or a virtual CPU.

Bits [31:24], ARMv7 base architecture

Reserved, RAZ.

Bits [31], Multiprocessing Extensions

RAO. Indicates that the processor implements the Multiprocessing Extensions register format.

U bit, bit [30], Multiprocessing Extensions

Indicates a Uniprocessor system, as distinct from processor 0 in a multiprocessor system. The possible values of this bit are:

0 Processor is part of a multiprocessor system.

1 Processor is part of a uniprocessor system

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.11 c0、マルチプロセッサ類似性レジスタ (MPIDR)

マルチプロセッサ類似性レジスタ (MPIDR) は、マルチプロセッサシステム内のスケジューリング用の補助的なプロセッサ識別機構を提供します。ユニプロセッサシステムでは、このレジスタから 0 の値を返すことをお勧めします。

MPIDR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、共通レジスタです。
- ARMv7 で導入されました。

ARMv7 基本アーキテクチャにおける MPIDR のフォーマットは次のとおりです。

マルチプロセッシング拡張機能が実装されている場合の MPIDR のフォーマットは次のとおりです。

注

MIDR のビット定義では、*システム内のプロセッサ*を物理プロセッサまたは仮想 CPU にすることができます。

ビット [31:24]、ARMv7 基本アーキテクチャ

予約、RAZ

ビット [31]、マルチプロセッシング拡張機能

RAO。プロセッサにマルチプロセッシング拡張機能レジスタフォーマットが実装されていることを示します。

U ビット（ビット [30]）、マルチプロセッシング拡張機能

マルチプロセッサシステムのプロセッサ 0 ではなく、ユニプロセッサシステムであることを示します。このビットの可能な値は次のとおりです。

0 プロセッサはマルチプロセッサシステムの一部です。

1 プロセッサはユニプロセッサシステムの一部です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Bits [29:25], Multiprocessing Extensions

Reserved, UNK.

MT bit, bit [24], Multiprocessing Extensions

Indicates whether the lowest level of affinity consists of logical processors that are implemented using a multi-threading type approach. The possible values of this bit are:

0 Performance of processors at the lowest affinity level is largely independent.

1 Performance of processors at the lowest affinity level is very interdependent

For more information about the meaning of this bit see *Multi-threading approach to lowest affinity levels, Multiprocessing Extensions* on page B3-89.

Affinity level 2, bits [23:16]

The least significant affinity level field, for this processor in the system.

Affinity level 1, bits [15:8]

The intermediate affinity level field, for this processor in the system.

Affinity level 0, bits [7:0]

The most significant level field, for this processor in the system.

In the system as a whole, for each of the affinity level fields, the assigned values must start at 0 and increase monotonically.

Increasing monotonically means that:

- There must not be any gaps in the sequence of numbers used.
- A higher value of the field includes any properties indicated by all lower values of the field.

When matching against an affinity level field, scheduler software checks for a value equal to or greater than a required value.

Recommended use of the MPIDR on page B3-89 includes a description of an example multiprocessor system and the affinity level field values it might use.

The interpretation of these fields is IMPLEMENTATION DEFINED, and must be documented as part of the documentation of the multiprocessor system. ARM recommends that this register might be used as described in the next subsection.

The software mechanism to discover the total number of affinity numbers used at each level is IMPLEMENTATION DEFINED, and is part of the general system identification task.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ビット [29:25]、マルチプロセッシング拡張機能

予約、UNK

MT ビット（ビット [24]）、マルチプロセッシング拡張機能

最低レベルの類似性が、マルチスレッド型の手法を使用して実装された論理プロセッサで構成されているかどうかを示します。このビットの可能な値は次のとおりです。

0 最低類似性レベルでのプロセッサのパフォーマンスが大部分独立している。

1 最低類似性レベルでのプロセッサのパフォーマンスが強く相互依存している。

このビットの意味の詳細については、*最低類似性レベルに対するマルチスレッド手法、マルチプロセッシング拡張機能* P. B3-89 を参照して下さい。

類似性レベル 2（ビット [23:16]）

最下位類似性レベルフィールド（システム内でこのプロセッサについて）

類似性レベル 1（ビット [15:8]）

中間類似性レベルフィールド（システム内でこのプロセッサについて）

類似性レベル 0（ビット [7:0]）

最上位類似性レベルフィールド（システム内でこのプロセッサについて）

システム全体で、類似性レベルフィールドごとに、割り当てる値は 0 から始まって単調増加する必要があります。

単調増加の意味は次のとおりです。

- 使用する数値の列に空白が存在しない。
- より高い値のフィールドには、より低い値のフィールドで示されるすべてのプロパティが含まれる。

類似性レベルフィールドに対する一致では、スケジューラソフトウェアによって必要な値以上かどうかがチェックされます。

マルチプロセッサシステムの例と、そのシステムで使用される可能性のある類似性レベルフィールドの値を、*MPIDR の推奨される使用法* P. B3-89 に示します。

これらのフィールドの解釈は実装定義で、マルチプロセッサシステムの説明書に記載する必要があります。このレジスタは、次のサブセクションで説明されているように使用することをお勧めします。

各レベルで使用される類似性番号の総数を検出するためのソフトウェア機構は実装定義で、一般的なシステム識別タスクの一部です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Multi-threading approach to lowest affinity levels, Multiprocessing Extensions

When the Multiprocessing Extensions are implemented, if the MPIDR.MT bit is set to 1, this indicates that the processors at affinity level 0 are logical processors, implemented using a multi-threading type approach. In such an approach, there can be a significant performance impact if a new thread is assigned the processor with:

- the same Affinity Level 0 value as some other thread, referred to as the original thread
- a pair of values for Affinity Levels 2 and 3 that are different to the pair of values of the original thread.

In this situation, the performance of the original thread might be significantly reduced.

Note

In this description, thread always refers to a thread or a process.

Recommended use of the MPIDR

In a multiprocessor system the register might provide two important functions:

- Identifying special functionality of a particular processor in the system. In general, the actual meaning of the affinity level fields is not important. In a small number of situations, an affinity level field value might have a special IMPLEMENTATION DEFINED significance. Possible examples include booting from reset and power-down events.
- Providing affinity information for the scheduling software, to help the scheduler run an individual thread or process on either:
  - the same processor, or as similar a processor as possible, as the processor it was running on previously
  - a processor on which a related thread or process was run.

Note

A monotonically increasing single number ID mechanism provides a convenient index into software arrays and for accessing the interrupt controller. This might be:

- performed as part of the boot sequence
- stored as part of the local storage of threads.

MPIDR provides a mechanism with up to three levels of affinity information, but the meaning of those levels of affinity is entirely IMPLEMENTATION DEFINED. The levels of affinity provided can have different meanings. Table B3-23 on page B3-90 shows two possible implementations:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

最低類似性レベルに対するマルチスレッド手法、マルチプロセッシング拡張機能

マルチプロセッシング拡張機能が実装されている場合、MPIDR.MT ビットが 1 にセットされていれば、類似性レベル 0 のプロセッサはマルチスレッド型の手法を使用して実装された論理プロセッサであることを示しています。このような手法では、新しいスレッドが次のようなプロセッサに割り当てられた場合に、パフォーマンスに大きく影響する可能性があります。

- 他のスレッド（元のスレッドと呼びます）と、類似性レベル 0 の値が同じである。
- 元のスレッドの値のペアと、類似性レベル 2 および 3 の値のペアが異なる。

この状況では、元のスレッドのパフォーマンスが大幅に低下する可能性があります。

注

この説明でのスレッドとは、常にスレッドまたはプロセスを意味します。

MPIDR の推奨される使用法

マルチプロセッサシステムでは、このレジスタによって次の 2 つの重要な機能が提供されます。

- システムの特定のプロセッサが持つ、特殊な機能の識別。一般的に、類似性レベルフィールドの実際の意味は重要ではありません。一部の状況では、類似性レベルフィールドの値が特殊な実装定義の意味を持つことがあります。この例としては、リセットや電力オフイベントからのブートが挙げられます。
- スケジューリングソフトウェアが、個別のスレッドまたはプロセスを次のようなプロセッサで実行を試みるときに役立てるため、スケジューラに類似性情報を提供する。
  - 以前にスレッドまたはプロセスを実行していたプロセッサと同じ、または可能な限り類似のプロセッサ
  - 関連するスレッドまたはプロセスが実行されていたプロセッサ

注

単調に増加する単一番号 ID 機構によって、ソフトウェアアレイと、割り込みコントローラへのアクセスにとって便利なインデクスが提供されます。この操作は、次の方法で実行できます。

- ブートシーケンスの一部として実行される。
- スレッドのローカル記憶域の一部としてストアされる。

MPIDR によって、最大で 3 レベルの類似性情報を持つ機構が提供されますが、これらのレベルの類似性の意味は、完全に実装定義です。提供する類似性のレベルによって意味を変えることができます。実現可能な 2 つの実装を、P. B3-90 の表 B3-23 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-23 Possible implementations of the affinity levels

Affinity Level	Example system 1	Example system 2
0	Virtual CPUs in a multi-threaded processor	Processors in an SMP cluster
1	Processors in an *Symmetric Multi Processor* (SMP) cluster	Clusters with a system
2	Clusters in a system	No meaning, fixed as 0.

The scheduler maintains affinity level information for all threads and processes. When it has to reschedule a thread or process the scheduler:

- looks for an available processor that matches at all three affinity levels
- if this fails, it might look for a processor that matches at levels 2 and 3 only
- if it still cannot find an available processor it might look for a match at level 3 only.

A multiprocessor system corresponding to Example system 1 in Table B3-23 might implement affinity values as shown in Table B3-24:

Table B3-24 Example of possible affinity values at different affinity levels

Affinity level 2 Cluster level	Affinity level 1 Processor level	Affinity level 0 Virtual CPU level
0	0	0, 1
0	1	0, 1
0	2	0, 1
0	3	0, 1
1	0	0, 1
1	1	0, 1
1	2	0, 1
1	3	0, 1

Accessing the MPIDR

To access MPIDR you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 5. For example:

```
MRC p15,0,<Rt>,c0,c0,5    ; Read Multiprocessor Affinity Register
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 B3-23 実現可能な類似性レベルの実装

類似性レベル	サンプルシステム 1	サンプルシステム 2
0	マルチスレッド プロセッサ内の仮想 CPU	SMP クラスタ内のプロセッサ
1	*対称型マルチプロセッサ*(SMP) クラスタ内のプロセッサ	システムを伴うクラスタ
2	システム内のクラスタ	意味なし、常に 0

スケジューラは、すべてのスレッドとプロセスに関する類似性レベル情報を保持します。スレッドまたはプロセスを再スケジュールする必要がある場合は、スケジューラが次の処理を実行します。

- 3 つすべての類似性レベルに一致する使用可能なプロセッサを探す。
- これに失敗した場合は、レベル 2 と 3 でのみ一致するプロセッサを探す。
- それでも使用可能なプロセッサが見つからない場合は、レベル 3 でのみ一致するプロセッサを探す。

表 B3-23 のサンプルシステム 1 に対応するマルチプロセッサシステムでは、表 B3-24 に示すような類似性の値を実装することができます。

表 B3-24 複数の類似性レベルで使用可能な類似性の値の例

類似性レベル 2 クラスタレベル	類似性レベル 1 プロセッサレベル	類似性レベル 0 仮想 CPU レベル
0	0	0, 1
0	1	0, 1
0	2	0, 1
0	3	0, 1
1	0	0, 1
1	1	0, 1
1	2	0, 1
1	3	0, 1

MPIDR へのアクセス

MPIDR にアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 5 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,0,<Rt>,c0,c0,5    ; Read Multiprocessor Affinity Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.12 c0, Cache Size ID Registers (CCSIDR)

The Cache Size ID Registers, CCSIDR, provide information about the architecture of the caches.

The CCSIDR registers are:

- 32-bit read-only registers
- accessible only in privileged modes
- when the Security Extensions are implemented, Common registers
- introduced in ARMv7.

One CCSIDR is implemented for each cache that can be accessed by the processor. CSSELR selects which Cache Size ID Register is accessible, see *c0, Cache Size Selection Register (CSSELR)* on page B3-95.

The format of a CCSIDR is:

31	30	29	28	27 … 13	12 … 3	2 … 0
WT	WB	RA	WA	NumSets	Associativity	LineSize

WT, bit [31] Indicates whether the cache level supports Write-Through, see Table B3-25.

WB, bit [30] Indicates whether the cache level supports Write-Back, see Table B3-25.

RA, bit [29] Indicates whether the cache level supports Read-Allocation, see Table B3-25.

WA, bit [28] Indicates whether the cache level supports Write-Allocation, see Table B3-25.

Table B3-25 WT, WB, RA and WA bit values

WT, WB, RA or WA bit value	Meaning
0	Feature not supported
1	Feature supported

NumSets, bits [27:13]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Associativity, bits [12:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]

(Log_2(Number of words in cache line)) -2. For example:

- For a line length of 4 words: $Log_2(4) = 2$, LineSize entry = 0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.12 c0、キャッシュサイズ ID レジスタ (CCSIDR)

キャッシュサイズ ID レジスタ (CCSIDR) は、キャッシュのアーキテクチャに関する情報を提供します。

CCSIDR レジスタの特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、共通レジスタです。
- ARMv7 で導入されました。

プロセッサからアクセス可能なキャッシュごとに、1 つの CCSIDR が実装されます。CSSELR によって、アクセス可能なキャッシュサイズ ID レジスタが選択されます。*c0、キャッシュサイズ選択レジスタ (CSSELR)* P. B3- 95 を参照して下さい。

CCSIDR のフォーマットは次のとおりです。

31	30	29	28	27 … 13	12 … 3	2 … 0
WT	WB	RA	WA	NumSets	Associativity	LineSize

WT、ビット [31] そのキャッシュレベルでライトスルーがサポートされているかどうかを示します。表 B3- 25 を参照して下さい。

WB、ビット [30] そのキャッシュレベルでライトバックがサポートされているかどうかを示します。表 B3- 25 を参照して下さい。

RA、ビット [29] そのキャッシュレベルで読み出し割り当てがサポートされているかどうかを示します。表 B3- 25 を参照して下さい。

WA、ビット [28] そのキャッシュレベルで書き込み割り当てがサポートされているかどうかを示します。表 B3- 25 を参照して下さい。

表 B3- 25 WT、WB、RA、WA ビットの値

WT、WB、RA、WA ビットの値	意味
0	機能がサポートされていない
1	機能がサポートされている

NumSets、ビット [27:13]

(キャッシュのセット数) − 1。したがって、0 の値はキャッシュが 1 セットであることを意味します。セット数を 2 の累乗にする必要はありません。

Associativity、ビット [12:3]

(キャッシュのアソシエティビティ) − 1。したがって、0 の値はアソシエティビティが 1 であることを意味します。アソシエティビティを 2 の累乗にする必要はありません。

LineSize、ビット [2:0]

(Log_2(キャッシュラインのワード数)) −2。次に例を示します。

- ライン長が 4 ワードの場合：$Log_2(4) = 2$、LineSize エントリ = 0

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

This is the minimum line length.

- For a line length of 8 words: $Log_2(8) = 3$, LineSize entry = 1.

Accessing the currently selected CCSIDR

The CSSELR selects a CCSIDR, see *c0, Cache Size Selection Register (CSSELR)* on page B3-95. To access the currently-selected CCSIDR you read the CP15 registers with `<opc1>` set to 1, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,1,<Rt>,c0,c0,0    ; Read current CP15 Cache Size ID Register
```

Accessing the CCSIDR when the value in CSSELR corresponds to a cache that is not implemented returns an UNKNOWN value.

B3.12.13 c0, Cache Level ID Register (CLIDR)

The Cache Level ID Register, CLIDR:

- identifies the type of cache, or caches, implemented at each level, up to a maximum of eight levels
- identifies the Level of Coherency and Level of Unification for the cache hierarchy.

The CLIDR is:

- a 32-bit read-only register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Common register
- introduced in ARMv7.

The format of the CLIDR is:

31	30	29 27	26 24	23 21	20 18	17 15	14 12	11 9	8 6	5 3	2 0
0	0	LoUU	LoC	LoUIS	Ctype7	Ctype6	Ctype5	Ctype4	Ctype3	Ctype2	Ctype1

Bits [31:30] RAZ.

LoUU, bits [29:27]

Level of Unification Uniprocessor for the cache hierarchy, see *Clean, Invalidate, and Clean and Invalidate* on page B2-11.

LoC, bits [26:24]

Level of Coherency for the cache hierarchy, see *Clean, Invalidate, and Clean and Invalidate* on page B2-11.

LoUIS, bits [23:21]

Level of Unification Inner Shareable for the cache hierarchy, see *Clean, Invalidate, and Clean and Invalidate* on page B2-11. This field is RAZ in implementations that do not implement the Multiprocessing Extensions.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

これは最小ライン長です。

- ライン長が 8 ワードの場合：$Log_2(8) = 3$、LineSize エントリ = 1

現在選択されている CCSIDR へのアクセス

CSSELR によって CCSIDR が選択されます。*c0、キャッシュサイズ選択レジスタ (CSSELR)* P. B3-95 を参照して下さい。現在選択されている CCSIDR にアクセスするには、`<opc1>` を 1、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,1,<Rt>,c0,c0,0    ; Read current CP15 Cache Size ID Register
```

CSSELR の値が、実装されていないキャッシュに対応している場合に、CCSIDR にアクセスすると、未知の値が返されます。

B3.12.13 c0、キャッシュレベル ID レジスタ (CLIDR)

キャッシュレベル ID レジスタ (CLIDR) の機能は次のとおりです。

- 最大 8 レベルのキャッシュについて、レベルごとに実装されているキャッシュタイプを識別する。
- キャッシュ階層のコヒーレンシレベルと統一レベルを識別する。

CLIDR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、共通レジスタです。
- ARMv7 で導入されました。

CLIDR のフォーマットは次のとおりです。

31	30	29 27	26 24	23 21	20 18	17 15	14 12	11 9	8 6	5 3	2 0
0	0	LoUU	LoC	LoUIS	Ctype7	Ctype6	Ctype5	Ctype4	Ctype3	Ctype2	Ctype1

ビット [31:30] RAZ

LoUU、ビット [29:27]

キャッシュ階層のユニプロセッサ統一レベル。*クリーニング、無効化、およびクリーニングと無効化* P. B2-11 を参照して下さい。

LoC、ビット [26:24]

キャッシュ階層のコヒーレンシレベル。*クリーニング、無効化、およびクリーニングと無効化* P. B2-11 を参照して下さい。

LoUIS、ビット [23:21]

キャッシュ階層の内部共有可能統一レベル。*クリーニング、無効化、およびクリーニングと無効化* P. B2-11 を参照して下さい。このフィールドは、マルチプロセッシング拡張機能を含まない実装では RAZ です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CtypeX, bits [3(x - 1) + 2:3(x - 1)], for x = 1 to 7

Cache Type fields. Indicate the type of cache implemented at each level, from Level 1 up to a maximum of seven levels of cache hierarchy. The Level 1 cache field, Ctype1, is bits [2:0], see register diagram. Table B3-26 shows the possible values for each CtypeX field.

Table B3-26 Ctype bit values

CtypeX value	Meaning, cache implemented at this level
000	No cache
001	Instruction cache only
010	Data cache only
011	Separate instruction and data caches
100	Unified cache
101, 11X	Reserved

If you read the Cache Type fields from Ctype1 upwards, once you have seen a value of 0b000, no caches exist at further out levels of the hierarchy. So, for example, if Ctype3 is the first Cache Type field with a value of 0b000, the values of Ctype4 to Ctype7 must be ignored.

The CLIDR describes only the caches that are under the control of the processor.

Accessing the CLIDR

To access the CLIDR you read the CP15 registers with `<opc1>` set to 1, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 1. For example:

```
MRC p15,1,<Rt>,c0,c0,1    ; Read CP15 Cache Level ID Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CtypeX、ビット [3(x − 1) + 2:3(x − 1)]、x = 1 ～ 7 の場合

キャッシュタイプ フィールド。キャッシュ階層のレベル 1 から、最大でレベル 7 までのキャッシュレベルごとに、実装されるキャッシュタイプを示します。レベル 1 キャッシュフィールド（Ctype1）は、ビット [2:0] です。レジスタ図を参照して下さい。各 CtypeX フィールドの可能な値を、表 B3-26 に示します。

表 B3-26 Ctype ビットの値

CtypeX の値	意味、このレベルに実装されるキャッシュ
000	キャッシュなし
001	命令キャッシュのみ
010	データキャッシュのみ
011	独立した命令キャッシュとデータキャッシュ
100	統一キャッシュ
101、11X	予約

Ctype1 から上に向かってキャッシュタイプ フィールドを読み出す場合は、0b000 の値を読み出した時点で、階層のその上のレベルにキャッシュが存在しないことを意味します。したがって、例えば、Ctype3 が 0b000 の値を持つ最初のキャッシュタイプ フィールドであれば、Ctype4 ～ Ctype7 の値を無視する必要があります。

CLIDR では、プロセッサの制御下にあるキャッシュのみが示されます。

CLIDR へのアクセス

CLIDR にアクセスするには、`<opc1>` を 1、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 1 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,1,<Rt>,c0,c0,1    ; Read CP15 Cache Level ID Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.14 c0, IMPLEMENTATION DEFINED Auxiliary ID Register (AIDR)

The IMPLEMENTATION DEFINED Auxiliary ID Register, AIDR, provides implementation-specific ID information. The value of this register must be used in conjunction with the value of MIDR.

The IMPLEMENTATION DEFINED AIDR is:

- a 32-bit read-only register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Common register
- introduced in ARMv7.

The format of the AIDR is IMPLEMENTATION DEFINED.

Accessing the AIDR

To access the AIDR you read the CP15 registers with `<opc1>` set to 1, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 7. For example:

```
MRC p15,1,<Rt>,c0,c0,7    ; Read IMPLEMENTATION DEFINED Auxiliary ID Register
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.12.14 c0、実装定義の補助 ID レジスタ (AIDR)

実装定義の補助 ID レジスタ (AIDR) は、実装固有の ID 情報を提供します。このレジスタの値は、MIDR の値と組み合わせて使用する必要があります。

実装定義の AIDR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、共通レジスタです。
- ARMv7 で導入されました。

AIDR のフォーマットは実装定義です。

AIDR へのアクセス

AIDR にアクセスするには、`<opc1>` を 1、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 7 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,1,<Rt>,c0,c0,7    ; Read IMPLEMENTATION DEFINED Auxiliary ID Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.15 c0, Cache Size Selection Register (CSSELR)

The Cache Size Selection Register, CSSELR, selects the current CCSIDR. An ARMv7 implementation must include a CCSIDR for every implemented cache that is under the control of the processor. The CSSELR identifies which CP1CSID register can be accessed, by specifying, for the required cache:

- the cache level
- the cache type, either:
  - — instruction cache.
  - — Data cache. The data cache argument is also used for a unified cache.

The CSSELR is:

- a 32-bit read/write register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Banked register
- introduced in ARMv7.

The format of the CSSELR is:

Bits [31:4] UNK/SBZP.

Level, bits [3:1]

Cache level of required cache. Permitted values are from 0b000, indicating Level 1 cache, to 0b110 indicating Level 7 cache.

InD, bit [0]

Instruction not Data bit. Permitted values are:

0 Data or unified cache

1 Instruction cache.

If CSSELR is set to indicate a cache that is not implemented, the result of reading CCSIDR is UNPREDICTABLE.

Accessing CSSELR

To access CSSELR you read or write the CP15 registers with `<opc1>` set to 2, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,2,<Rt>,c0,c0,0    ; Read Cache Size Selection Register
MCR p15,2,<Rt>,c0,c0,0    ; Write Cache Size Selection Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.15 c0、キャッシュサイズ選択レジスタ (CSSELR)

キャッシュサイズ選択レジスタ (CSSELR) は、現在の CCSIDR を選択します。ARMv7 実装には、プロセッサの制御下にある、実装されているすべてのキャッシュについて CCSIDR を含める必要があります。CSSELR では、必要なキャッシュについて次の要素を指定することによって、アクセス可能な CP1CSID レジスタが識別されます。

- キャッシュレベル
- 次のいずれかのキャッシュタイプ
  - – 命令キャッシュ
  - – データキャッシュ。データキャッシュの引数は、統一キャッシュにも使用されます。

CSSELR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、バンクレジスタです。
- ARMv7 で導入されました。

CSSELR のフォーマットは次のとおりです。

ビット [31:4] UNK/SBZP

レベル、ビット [3:1]

必要なキャッシュのキャッシュレベル。許される値は、レベル 1 キャッシュを示す 0b000 から、レベル 7 キャッシュを示す 0b110 までです。

InD、ビット [0]

命令 / データビット。許される値は次のとおりです。

0 データまたは統一キャッシュ

1 命令キャッシュ

CSSELR が実装されていないキャッシュを示すように設定されている場合、CCSIDR の読み出し結果は予測不能です。

CSSELR へのアクセス

CSSELR にアクセスするには、`<opc1>` を 2、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,2,<Rt>,c0,c0,0    ; Read Cache Size Selection Register
MCR p15,2,<Rt>,c0,c0,0    ; Write Cache Size Selection Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.16 CP15 c1, System control registers

The CP15 c1 registers are used for system control. Figure B3-12 shows the CP15 c1 registers.

Figure B3-12 CP15 c1 registers in a VMSA implementation

CP15 c1 register encodings not shown in Figure B3-12 are UNPREDICTABLE. When the Security Extensions are not implemented all encodings with CRm == c1 are UNPREDICTABLE. For more information, see *Unallocated CP15 encodings* on page B3-69.

The following sections describe the CP15 c1 registers:

- *c1, System Control Register (SCTLR)*
- *c1, Implementation defined Auxiliary Control Register (ACTLR)* on page B3-103
- *c1, Coprocessor Access Control Register (CPACR)* on page B3-104
- *c1, Secure Configuration Register (SCR)* on page B3-106
- *c1, Secure Debug Enable Register (SDER)* on page B3-108
- *c1, Non-Secure Access Control Register (NSACR)* on page B3-110.

B3.12.17 c1, System Control Register (SCTLR)

The System Control Register, SCTLR, provides the top level control of the system, including its memory system.

The SCTLR:

- Is a 32-bit read/write register, with different access rights for some bits of the register.

  In ARMv7, some bits in the register are read-only. These bits relate to non-configurable features of an ARMv7 implementation, and are provided for compatibility with previous versions of the architecture.

- Is accessible only in privileged modes.

- Has a defined reset value. The reset value is IMPLEMENTATION DEFINED, see *Reset value of the SCTLR* on page B3-102. When the Security Extensions are implemented the defined reset value applies only to the Secure copy of the SCTLR, and software must program the non-banked read/write bits of the Non-secure copy of the register with the required values.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.16 CP15 c1、システム制御レジスタ

CP15 c1 レジスタは、システム制御に使用されます。CP15 c1 レジスタを、図 B3-12 に示します。

図 B3-12 VMSA 実装の CP15 c1 レジスタ

図 B3-12 に示されていない CP15 c1 レジスタのエンコードは、予測不能です。セキュリティ拡張機能が実装されていない場合は、CRm == c1 を使用したすべてのエンコードが予測不能です。詳細については、*未割り当ての CP15 エンコード* P. B3-69 を参照して下さい。

CP15 c1 レジスタについては、次のセクションで説明します。

- *c1、システム制御レジスタ (SCTLR)*
- *c1、実装定義の補助制御レジスタ (ACTLR)* P. B3-103
- *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B3-104
- *c1、セキュア構成レジスタ (SCR)* P. B3-106
- *c1、セキュアデバッグ イネーブルレジスタ (SDER)* P. B3-108
- *c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3-110

B3.12.17 c1、システム制御レジスタ (SCTLR)

システム制御レジスタ (SCTLR) は、メモリシステムを含むシステムのトップレベルの制御を提供します。

SCTLR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタで、一部のビットはアクセス権限が異なっています。

  ARMv7 では、このレジスタの一部のビットが読み出し専用です。これらのビットは、ARMv7 実装の構成不可機能に関係しており、以前のバージョンのアーキテクチャとの互換性を維持するために提供されています。

- 特権モードでのみアクセス可能

- リセット時の値が定義されている。リセット時の値は実装定義です。*SCTLR のリセット値* P. B3-102 を参照して下さい。セキュリティ拡張機能が実装されている場合は、定義されたリセット時の値は SCTLR のセキュアコピーにのみ適用されるため、ソフトウェアでレジスタの非セキュアコピーのバンクされない読み出し / 書き込みビットを必要な値にプログラムする必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- When the Security Extensions are implemented:
  - — is a Banked register, with some bits common to the Secure and Non-secure copies of the register
  - — has write access to the Secure copy of the register disabled when the **CP15SDISABLE** signal is asserted HIGH.

  For more information, see *Effect of the Security Extensions on the CP15 registers* on page B3-71.

Control bits in the SCTLR that are not applicable to a VMSA implementation read as the value that most closely reflects that implementation, and ignore writes.

In an ARMv7-A implementation the format of the SCTLR is:

Bit [31] UNK/SBZP.

TE, bit [30] Thumb Exception enable. This bit controls whether exceptions are taken in ARM or Thumb state:

0 Exceptions, including reset, handled in ARM state

1 Exceptions, including reset, handled in Thumb state.

When the Security Extensions are implemented, this bit is banked between the Secure and Non-secure versions of the register.

An implementation can include a configuration input signal that determines the reset value of the TE bit. If there is no configuration input signal to determine the reset value of this bit then it resets to 0 in an ARMv7-A implementation.

For more information about the use of this bit see *Instruction set state on exception entry* on page B1-35.

AFE, bit [29] Access Flag Enable bit. This bit enables use of the AP[0] bit in the translation table descriptors as an *access flag*. It also restricts access permissions in the translation table descriptors to the simplified model described in *Simplified access permissions model* on page B3-29. The possible values of this bit are:

0 In the translation table descriptors, AP[0] is an access permissions bit. The full range of access permissions is supported. No access flag is implemented.

1 In the translation table descriptors, AP[0] is an access flag. Only the simplified model for access permissions is supported.

When the Security Extensions are implemented, this bit is banked between the Secure and Non-secure versions of the register.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- セキュリティ拡張機能が実装されている場合、次の特徴があります。
  - – バンクレジスタで、一部のビットがレジスタのセキュアコピーと非セキュアコピーで共通です。
  - – レジスタのセキュアコピーへの書き込みアクセスは、CP15SDISABLE 信号が HIGH にアサートされていたときには不可能です。

  詳細については、*CP15 レジスタに対するセキュリティ拡張機能の影響* P. B3-71 を参照して下さい。

VMSA 実装に適用されない SCTLR の制御ビットは、その実装を最も的確に反映している値として読み出されます。書き込みは無視されます。

ARMv7-A 実装における SCTLR のフォーマットは次のとおりです。

ビット [31] UNK/SBZP

TE、ビット [30] Thumb 例外イネーブル。このビットによって、ARM 状態と Thumb 状態のどちらで例外が取得されるかが制御されます。

0 例外は、リセットも含めて、ARM 状態で処理されます。

1 例外は、リセットも含めて、Thumb 状態で処理されます。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。

実装には、TE ビットのリセット時の値を決定する構成入力信号を含めることができます。ARMv7-A では、このビットのリセット時の値を決定するための構成入力信号が存在しない場合は、このビットが 0 にリセットされます。

このビットの使用方法については、*例外の開始時点の命令セット状態* P. B1-35 を参照して下さい。

AFE、ビット [29] アクセスフラグ イネーブルビット。このビットによって、変換テーブル記述子の AP[0] ビットを *アクセスフラグ*として使用することができます。また、変換テーブル記述子でのアクセス許可は、*単純なアクセス許可モデル* P. B3-29 で説明されている単純化されたモデルに制限されます。このビットの可能な値は次のとおりです。

0 変換テーブル記述子の AP[0] がアクセス許可ビットです。アクセス許可の全範囲がサポートされます。アクセスフラグは実装されません。

1 変換テーブル記述子の AP[0] がアクセスフラグです。アクセス許可の単純化されたモデルのみがサポートされます。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

TRE, bit [28] TEX Remap Enable bit. This bit enables remapping of the TEX[2:1] bits for use as two translation table bits that can be managed by the operating system. Enabling this remapping also changes the scheme used to describe the memory region attributes in the VMSA. The possible values of this bit are:

0 TEX Remap disabled. TEX[2:0] are used, with the C and B bits, to describe the memory region attributes.

1 TEX Remap enabled. TEX[2:1] are reassigned for use as flags managed by the operating system. The TEX[0], C and B bits are used to describe the memory region attributes, with the MMU remap registers.

When the Security Extensions are implemented, this bit is banked between the Secure and Non-secure versions of the register.

For more information, see *The alternative descriptions of the Memory region attributes* on page B3-32.

NMFI, bit [27]

Non-maskable Fast Interrupts enable:

0 Fast interrupts (FIQs) can be masked in the CPSR

1 Fast interrupts are non-maskable.

When the Security Extensions are implemented this bit is common to the Secure and Non-secure versions of the register.

This bit is read-only. It is IMPLEMENTATION DEFINED whether an implementation supports *Non-Maskable Fast Interrupts* (NMFIs):

- If NMFIs are not supported then this bit must be RAZ.
- If NMFIs are supported then this bit is controlled by a configuration input signal.

For more information, see *Non-maskable fast interrupts* on page B1-18.

Bit [26] RAZ/SBZP.

EE, bit [25] Exception Endianness bit. The value of this bit defines the value of the CPSR.E bit on entry to an exception vector, including reset. This value also indicates the endianness of the translation table data for translation table lookups. The permitted values of this bit are:

0 Little endian

1 Big endian.

When the Security Extensions are implemented, this bit is banked between the Secure and Non-secure versions of the register.

This is a read/write bit. An implementation can include a configuration input signal that determines the reset value of the EE bit. If there is no configuration input signal to determine the reset value of this bit then it resets to 0.

VE, bit [24] Interrupt Vectors Enable bit. This bit controls the vectors used for the FIQ and IRQ interrupts. The permitted values of this bit are:

0 Use the FIQ and IRQ vectors from the vector table, see the V bit entry

1 Use the IMPLEMENTATION DEFINED values for the FIQ and IRQ vectors.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

TRE、ビット[28] TEX 再マップイネーブル ビット。このビットによって、TEX[2:1] ビットの再マッピングを、オペレーティングシステムで管理可能な 2 つの変換テーブルビットとして使用することができます。この再マッピングを有効にすることによって、VMSA のメモリ領域属性を示すために使用される方式も変更されます。このビットの可能な値は次のとおりです。

0 TEX 再マップ禁止。C ビット、B ビットとともに TEX[2:0] を使用して、メモリ領域の属性が記述されます。

1 TEX 再マップ許可。TEX[2:1] が、オペレーティングシステムで管理されるフラグとして再割り当てされます。MMU 再マップレジスタとともに、TEX[0]、C、B ビットを使用してメモリ領域属性が記述されます。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。

詳細については、*メモリ領域の属性の代替表記* P. B3- 32 を参照して下さい。

NMFI、ビット [27]

マスク不能高速割り込みイネーブル

0 CPSR で高速割り込み (FIQ) をマスクできます。

1 高速割り込みはマスク不可です。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとで共通です。

このビットは読み出し専用です。実装で *マスク不能高速割り込み* (NMFI) がサポートされるかどうかは、実装定義です。

- NMFI がサポートされていない場合は、このビットを RAZ にする必要があります。
- NMFI がサポートされている場合、このビットは構成入力信号によって制御されます。

詳細については、*マスク不能高速割り込み* P. B1- 18 を参照して下さい。

ビット [26] RAZ/SBZP

EE、ビット[25] 例外エンディアン形式ビット。このビットの値によって、リセットを含む例外ベクタのエントリでの CPSR.E ビットの値が定義されます。また、変換テーブルルックアップ用の変換テーブルデータのエンディアン形式も示されます。このビットに許される値は次のとおりです。

0 リトルエンディアン

1 ビッグエンディアン

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。

読み出し / 書き込みビットです。実装には、EE ビットのリセット時の値を決定する構成入力信号を含めることができます。リセット時の値を決定する構成入力信号がない場合、このビットのリセット時の値は 0 です。

VE、ビット[24] 割り込みベクタイネーブル ビット。このビットによって、FIQ 割り込みと IRQ 割り込みで使用されるベクタが制御されます。このビットに許される値は次のとおりです。

0 ベクタテーブルから FIQ ベクタと IRQ ベクタを使用します（V ビットのエントリを参照）。

1 FIQ ベクタと IRQ ベクタの実装定義の値を使用します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When the Security Extensions are implemented, this bit is banked between the Secure and Non-secure versions of the register.

For more information, see *Vectored interrupt support* on page B1-32.

If the implementation does not support IMPLEMENTATION DEFINED FIQ and IRQ vectors then this bit is RAZ/WI.

Bit [23] RAO/SBOP.

U, bit [22] In ARMv7 this bit is RAO/SBOP, indicating use of the alignment model described in *Alignment support* on page A3-4.

For details of this bit in earlier versions of the architecture see *Alignment* on page AppxG-6.

FI, bit [21] Fast Interrupts configuration enable bit. This bit can be used to reduce interrupt latency in an implementation by disabling IMPLEMENTATION DEFINED performance features. The permitted values of this bit are:

0 All performance features enabled.

1 Low interrupt latency configuration. Some performance features disabled.

When the Security Extensions are implemented, this bit is common to the Secure and Non-secure versions of the register.

This bit is:

- a read/write bit if the Security Extensions are not implemented
- if the Security Extensions are implemented:
  - — a read/write bit if the processor is in Secure state
  - — a read-only bit if the processor is in Non-secure state.

For more information, see *Low interrupt latency configuration* on page B1-43.

If the implementation does not support a mechanism for selecting a low interrupt latency configuration this bit is RAZ/WI.

Bit [20:19] RAZ/SBZP.

Bit [18] RAO/SBOP.

HA, bit [17] Hardware Access Flag Enable bit. If the implementation provides hardware management of the access flag this bit enables the access flag management:

0 Hardware management of access flag disabled

1 Hardware management of access flag enabled.

If the Security Extensions are implemented then this bit is banked between the Secure and Non-secure versions of the register.

If the implementation does not provide hardware management of the access flag then this bit is RAZ/WI.

For more information, see *Hardware management of the access flag* on page B3-21.

Bit [16] RAO/SBOP.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。

詳細については、*ベクタ割り込みのサポート* P. B1-32 を参照して下さい。

実装で実装定義の FIQ ベクタと IRQ ベクタがサポートされていない場合、このビットは RAZ/WI です。

ビット [23] RAO/SBOP

U、ビット[22] ARMv7 では、このビットは RAO/SBOP で、*アライメントのサポート* P. A3-4 で説明されているアライメントモデルが使用されることを示します。

以前のバージョンのアーキテクチャにおけるこのビットの詳細については、*アライメント* P. 付録 G-6 を参照して下さい。

FI、ビット[21] 高速割り込み構成イネーブルビット。このビットは、実装定義のパフォーマンス機能を禁止することによって、実装内の割り込みレイテンシを低減するために使用できます。このビットに許される値は次のとおりです。

0 すべてのパフォーマンス機能を許可します。

1 低割り込みレイテンシ構成。一部のパフォーマンス機能を禁止します。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとで共通です。

このビットの特徴は次のとおりです。

- セキュリティ拡張機能が実装されていない場合は、読み出し / 書き込みビット
- セキュリティ拡張機能が実装されている場合は、次のようになります。
  - – プロセッサがセキュア状態の場合、読み出し / 書き込みビット
  - – プロセッサが非セキュア状態の場合、読み出し専用ビット

詳細については、*低割り込みレイテンシ構成* P. B1-43 を参照して下さい。

実装で低割り込みレイテンシ構成の選択機構がサポートされていない場合、このビットは RAZ/WI です。

ビット [20:19] RAZ/SBZP

ビット [18] RAO/SBOP

HA、ビット[17] ハードウェアアクセスフラグ イネーブルビット。実装でアクセスフラグのハードウェア管理が提供されている場合、このビットによってアクセスフラグ管理が許可されます。

0 アクセスフラグのハードウェア管理は不可能です。

1 アクセスフラグのハードウェア管理が可能です。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。

実装でアクセスフラグのハードウェア管理が提供されない場合、このビットは RAZ/WI です。

詳細については、*アクセスフラグのハードウェア管理* P. B3-22 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Bit [15] RAZ/SBZP.

RR, bit [14] Round Robin bit. If the cache implementation supports the use of an alternative replacement strategy that has a more easily predictable worst-case performance, this bit selects it:

0 Normal replacement strategy, for example, random replacement

1 Predictable strategy, for example, round-robin replacement.

When the Security Extensions are implemented, this bit is common to the Secure and Non-secure versions of the register.

This bit is:

- a read/write bit if the Security Extensions are not implemented
- if the Security Extensions are implemented:
  - a read/write bit if the processor is in Secure state
  - a read-only bit if the processor is in Non-secure state.

The replacement strategy associated with each value of the RR bit is IMPLEMENTATION DEFINED.

If the implementation does not support multiple IMPLEMENTATION DEFINED replacement strategies this bit is RAZ/WI.

V, bit [13] Vectors bit. This bit selects the base address of the exception vectors:

0 Normal exception vectors, base address `0x00000000`.

When the Security Extensions are implemented this base address can be re-mapped.

1 High exception vectors (Hivecs), base address `0xFFFF0000`.

This base address is never remapped.

When the Security Extensions are implemented, this bit is banked between the Secure and Non-secure versions of the register.

An implementation can include a configuration input signal that determines the reset value of the V bit. If there is no configuration input signal to determine the reset value of this bit then it resets to 0.

For more information, see *Exception vectors and the exception base address* on page B1-30.

I, bit [12] Instruction cache enable bit: This is a global enable bit for instruction caches:

0 Instruction caches disabled

1 Instruction caches enabled.

When the Security Extensions are implemented, this bit is banked between the Secure and Non-secure versions of the register.

If the system does not implement any instruction caches that can be accessed by the processor, at any level of the memory hierarchy, this bit is RAZ/WI.

If the system implements any instruction caches that can be accessed by the processor then it must be possible to disable them by setting this bit to 0.

Cache enabling and disabling on page B2-8 describes the effect of enabling the caches.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ビット [16] RAO/SBOP

ビット [15] RAZ/SBZP

RR、ビット[14] ラウンドロビン ビット。キャッシュ実装で、ワーストケースのパフォーマンスの予測を容易にする代替置換方式の使用がサポートされている場合は、このビットによって次の手法が選択されます。

0 ランダム置換などの通常の置換方式

1 ラウンドロビン置換などの予測可能な置換方式

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとで共通です。

このビットの特徴は次のとおりです。

- セキュリティ拡張機能が実装されていない場合は、読み出し / 書き込みビット
- セキュリティ拡張機能が実装されている場合は、次のようになります。
  - プロセッサがセキュア状態の場合、読み出し / 書き込みビット
  - プロセッサが非セキュア状態の場合、読み出し専用ビット

RR ビットのそれぞれの値に関連付けられている置換方式は、実装定義です。

実装で複数の実装定義の置換方式がサポートされていない場合、このビットは RAZ/WI です。

V、ビット[13] ベクタビット。このビットにより、次のように例外ベクタのベースアドレスが選択されます。

0 通常の例外ベクタ、ベースアドレス `0x00000000`。

セキュリティ拡張機能が実装されている場合、このベースアドレスを再マッピングすることができます。

1 上位例外ベクタ (Hivecs)、ベースアドレス `0xFFFF0000`。

このベースアドレスは再マッピングされることはありません。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。

実装には、V ビットのリセット時の値を決定する構成入力信号を含めることができます。リセット時の値を決定する構成入力信号がない場合、このビットのリセット時の値は 0 です。

詳細については、*例外ベクタと例外ベースアドレス* P. B1-30 を参照して下さい。

I、ビット[12] 命令キャッシュイネーブル ビット。命令キャッシュのグローバルイネーブル ビットです。

0 命令キャッシュを禁止します。

1 命令キャッシュを許可します。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。

システムのメモリ階層のどのレベルにも、プロセッサからアクセス可能な命令キャッシュが実装されていない場合、このビットは RAZ/WI です。

システムにプロセッサからアクセス可能な命令キャッシュが実装されている場合は、このビットを 0 にセットしてそれらのキャッシュを禁止可能な必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Z, bit [11] Branch prediction enable bit. This bit is used to enable branch prediction, also called program flow prediction:

0 Program flow prediction disabled

1 Program flow prediction enabled.

When the Security Extensions are implemented, this bit is banked between the Secure and Non-secure versions of the register.

If program flow prediction cannot be disabled, this bit is RAO/WI. Program flow prediction includes all possible forms of speculative change of instruction stream prediction. Examples include static prediction, dynamic prediction, and return stacks.

If the implementation does not support program flow prediction this bit is RAZ/WI.

SW, bit[10] SWP/SWPB Enable bit. This bit enables the use of `SWP` and `SWPB` instructions:

0 `SWP` and `SWPB` are UNDEFINED

1 `SWP` and `SWPB` perform as described in *SWP, SWPB* on page A8-432.

When the Security Extensions are implemented, this bit is banked between the Secure and Non-secure versions of the register. The bit is reset to 0.

This is part of the Multiprocessing Extensions. In implementations that do not implement the Multiprocessing Extensions this bit is RAZ and `SWP` and `SWPB` instructions perform as described in *SWP, SWPB* on page A8-432.

Note

At reset, this bit disables `SWP` and `SWPB`. This means that operating systems have to choose to use `SWP` or `SWPB`.

Bits [9:8] RAZ/SBZP.

B, bit [7] In ARMv7 this bit is RAZ/SBZP, indicating use of the endianness model described in *Endian support* on page A3-7.

For details of this bit in earlier versions of the architecture see *Endian support* on page AppxG-7 and *Endian support* on page AppxH-7.

Bits [6:3] RAO/SBOP.

C, bit [2] Cache enable bit: This is a global enable bit for data and unified caches:

0 Data and unified caches disabled

1 Data and unified caches enabled.

When the Security Extensions are implemented, this bit is banked between the Secure and Non-secure versions of the register.

If the system does not implement any data or unified caches that can be accessed by the processor, at any level of the memory hierarchy, this bit is RAZ/WI.

If the system implements any data or unified caches that can be accessed by the processor then it must be possible to disable them by setting this bit to 0.

Cache enabling and disabling on page B2-8 describes the effect of enabling the caches.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

キャッシュの稼動の影響については、*キャッシュの稼動と非稼動* P. B2-8 で説明します。

Z、ビット[11] 分岐予測イネーブルビット。このビットは分岐予測（プログラムフロー予測とも呼ばれます）を許可するために使用されます。

0 プログラムフロー予測は非稼動です。

1 プログラムフロー予測が可能です。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。

プログラムフロー予測を禁止することができない場合、このビットは RAO/WI です。プログラムフロー予測には、命令ストリーム予測の投機的変更の使用可能な形式がすべて含まれます。例としては、静的予測、動的予測、リターンスタックなどがあります。

実装でプログラムフロー予測がサポートされていない場合、このビットは RAZ/WI です。

SW、ビット[10] SWP/SWPB イネーブルビット。このビットは、`SWP` 命令と `SWPB` 命令の使用を許可します。

0 `SWP` と `SWPB` は未定義です。

1 `SWP` と `SWPB` が、*SWP*、*SWPB* P. A8-432 で説明されているように動作します。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。このビットは 0 にリセットされます。

これは、マルチプロセッシング拡張機能の一部です。マルチプロセッシング拡張機能を含まない実装では、このビットは RAZ で、`SWP` 命令と `SWPB` 命令は *SWP*、*SWPB* P. A8-432 で説明されているように動作します。

注

リセット時には、このビットによって `SWP` と `SWPB` が禁止されます。これは、オペレーティングシステムで `SWP` または `SWPB` の使用を選択する必要があることを意味します。

ビット [9:8] RAZ/SBZP

B、ビット [7] ARMv7 では、このビットは RAZ/SBZP で、*エンディアンのサポート* P. A3-7 で説明されているエンディアン形式モデルの使用を示します。

以前のバージョンのアーキテクチャにおけるこのビットの詳細については、*エンディアンのサポート* P. 付録 G-7 と*エンディアンのサポート* P. 付録 H-7 を参照して下さい。

ビット [6:3] RAO/SBOP

C、ビット [2] キャッシュイネーブル ビット。データキャッシュおよび統一キャッシュのグローバルイネーブル ビットです。

0 データキャッシュおよび統一キャッシュは非稼動です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A, bit [1] Alignment bit. This is the enable bit for Alignment fault checking:

0 Alignment fault checking disabled

1 Alignment fault checking enabled.

When the Security Extensions are implemented, this bit is banked between the Secure and Non-secure versions of the register.

For more information, see *Alignment fault* on page B3-42, for a VMSA implementation.

M, bit [0] MMU enable bit. This is a global enable bit for the MMU:

0 MMU disabled

1 MMU enabled.

When the Security Extensions are implemented, this bit is banked between the Secure and Non-secure versions of the register.

For more information, see *Enabling and disabling the MMU* on page B3-5.

Reset value of the SCTLR

The SCTLR has a defined reset value that is IMPLEMENTATION DEFINED. There are different types of bit in the SCTLR:

- Some bits are defined as RAZ or RAO, and have the same value in all VMSAv7 implementations. Figure B3-13 on page B3-103 shows the values of these bits.
- Some bits are read-only and either:
  - have an IMPLEMENTATION DEFINED value
  - have a value that is determined by a configuration input signal.
- Some bits are read/write and either:
  - reset to zero
  - reset to an IMPLEMENTATION DEFINED value
  - reset to a value that is determined by a configuration input signal.

Figure B3-13 on page B3-103 shows the reset value, or how the reset value is defined, for each bit of the SCTLR. It also shows the possible values of each half byte of the register.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

1 データキャッシュおよび統一キャッシュが稼動中です。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。

システムのメモリ階層のどのレベルにも、プロセッサからアクセス可能なデータまたは統一キャッシュが実装されていない場合、このビットは RAZ/WI です。

システムに、プロセッサからアクセス可能なデータまたは統一キャッシュが実装されている場合は、このビットを 0 にセットして、それらのキャッシュを非稼動にできる必要があります。

キャッシュの稼動の影響については、*キャッシュの稼動と非稼動* P. B2-8 で説明します。

A、ビット[1] アライメントビット。アライメントフォールト チェックのイネーブルビットです。

0 アライメントフォールト チェックを不可能にします。

1 アライメントフォールト チェックを可能にします。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。

VMSA 実装の場合の詳細については、*アライメントフォールト* P. B3-42 を参照して下さい。

M、ビット[0] MMU イネーブルビット。MMU のグローバルイネーブル ビットです。

0 MMU は非稼動です。

1 MMU が稼動中です。

セキュリティ拡張機能が実装されている場合、このビットはレジスタのセキュアバージョンと非セキュアバージョンとの間でバンクされます。

詳細については、*MMU の許可と禁止* P. B3-5 を参照して下さい。

SCTLR のリセット値

SCTLR のリセット時の値は実装定義です。SCTLR にはさまざまな種類のビットがあります。

- 一部のビットは RAZ または RAO として定義され、すべての VMSAv7 実装で同じ値です。これらのビットの値を、P. B3-103 の図 B3-13 に示します。
- 一部のビットは読み出し専用で、次のいずれかです。
  - 実装定義の値を持つ。
  - 構成入力信号によって決定される値を持つ。
- 一部のビットは読み出し / 書き込み可能で、次のいずれかです。
  - 0 にリセットされる。
  - 実装定義の値にリセットされる。
  - 構成入力信号によって決定される値にリセットされる。

SCTLR の各ビットのリセット時の値、またはリセット時の値がどのように定義されるかを、P. B3-103 の図 B3-13 に示します。レジスタの各ハーフバイトの可能な値も示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

* Read-only bits, including RAZ and RAO bits.
(*) Can be RAZ. Otherwise read/write, resets to 0.
(†) Can be read-only, with IMPLEMENTATION DEFINED value. Otherwise resets to 0.
‡ Value or reset value can depend on configuration input. Otherwise RAZ or resets to 0.

Figure B3-13 Reset value of the SCTLR, ARMv7-A (VMSAv7)

Accessing the SCTLR

To access the SCTLR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c1, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c1,c0,0    ; Read CP15 System Control Register
MCR p15,0,<Rt>,c1,c0,0    ; Write CP15 System Control Register
```

Note

Additional configuration and control bits might be added to the SCTLR in future versions of the ARM architecture. ARM strongly recommends that software always uses a read, modify, write sequence to update the SCTLR. This prevents software modifying any bit that is currently unallocated, and minimizes the chance of the register update having undesired side effects.

B3.12.18 c1, IMPLEMENTATION DEFINED Auxiliary Control Register (ACTLR)

The Auxiliary Control Register, ACTLR, provides implementation-specific configuration and control options.

The ACTLR is:

- A 32-bit read/write register.
- Accessible only in privileged modes.
- When the Security Extensions are implemented, a Banked register. However, some bits might define global configuration settings, and be common to the Secure and Non-secure copies of the register.

The contents of this register are IMPLEMENTATION DEFINED. ARMv7 requires this register to be privileged read/write accessible, even if an implementation has not created any control bits in this register.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

* RAZビットとRAOビットを含む読み出し専用ビット
(*) RAZにすることができる。それ以外は、読み出し/ 書き込み、0にリセット。
(†) 実装定義の値を使用した読み出し専用にすることができる。それ以外は、0にリセット。
‡ 値またはリセット値は構成入力によって変えることができる。
それ以外は、RAZまたは0にリセット。

図 B3- 13 SCTLR のリセット時の値、ARMv7-A (VMSAv7)

SCTLR へのアクセス

SCTLR にアクセスするには、`<opc1>` を 0、`<CRn>` を c1、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c1,c0,0    ; Read CP15 System Control Register
MCR p15,0,<Rt>,c1,c0,0    ; Write CP15 System Control Register
```

注

将来のバージョンの ARM アーキテクチャで、新しい構成ビットと制御ビットが SCTLR に追加される可能性があります。ソフトウェアで SCTLR を更新する場合は、常に読み出し – 変更 – 書き込みシーケンスを使用することを強くお勧めします。この方法を使用すれば、現時点で割り当てられていないビットがソフトウェアで変更されないため、不要な副作用を伴うレジスタ更新の機会が最小化されます。

B3.12.18 c1、実装定義の補助制御レジスタ (ACTLR)

補助制御レジスタ (ACTLR) は、実装固有の構成および制御のオプションを提供します。

ACTLR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、バンクレジスタです。ただし、一部のビットはグローバル構成設定の定義に使用され、レジスタのセキュアコピーと非セキュアコピーとで共通の場合があります。

このレジスタの内容は実装定義です。ARMv7 では、実装でこのレジスタにいずれの制御ビットも作成されていない場合でも、このレジスタを特権読み出し / 書き込みアクセス可能にする必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the ACTLR

To access the ACTLR you read or write the CP15 registers with <opc1> set to 0, <CRn> set to c1, <CRm> set to c0, and <opc2> set to 1. For example:

```
MRC p15,0,<Rt>,c1,c0,1    ; Read CP15 Auxiliary Control Register
MCR p15,0,<Rt>,c1,c0,1    ; Write CP15 Auxiliary Control Register
```

B3.12.19 c1, Coprocessor Access Control Register (CPACR)

The Coprocessor Access Control Register, CPACR, controls access to all coprocessors other than CP14 and CP15. It also enables software to check for the presence of coprocessors CP0 to CP13.

The CPACR:

- is a 32-bit read/write register
- is accessible only in privileged modes
- has a defined reset value of 0
- when the Security Extensions are implemented, is a Configurable access register.

The format of the CPACR is:

ASEDIS, bit[31]

Disable Advanced SIMD Functionality:

0 This bit does not cause any instructions to be UNDEFINED.

1 All instruction encodings identified in the *Alphabetical list of instructions* on page A8-14 as being part of Advanced SIMD, but that are not VFPv3 instructions, are UNDEFINED.

On an implementation that:

- Implements VFP and does not implement Advanced SIMD, this bit is RAO/WI.
- Does not implement VFP or Advanced SIMD, this bit is UNK/SBZP.
- Implements both VFP and Advanced SIMD, it is IMPLEMENTATION DEFINED whether this bit is supported. If it is not supported it is RAZ/WI.

This bit resets to 0 if it is supported.

D32DIS, bit[30]

Disable use of D16-D31 of the VFP register file:

0 This bit does not cause any instructions to be UNDEFINED.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ACTLR へのアクセス

ACTLR にアクセスするには、<opc1> を 0、<CRn> を c1、<CRm> を c0、<opc2> を 1 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c1,c0,1    ; Read CP15 Auxiliary Control Register
MCR p15,0,<Rt>,c1,c0,1    ; Write CP15 Auxiliary Control Register
```

B3.12.19 c1、コプロセッサアクセス制御レジスタ (CPACR)

コプロセッサアクセス制御レジスタ (CPACR) は、CP14 と CP15 を除くすべてのコプロセッサへのアクセスを制御します。また、ソフトウェアでコプロセッサの CP0 ～ CP13 の存在のチェックを可能にします。

CPACR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- リセット時の値は 0 に定義されています。
- セキュリティ拡張機能が実装されている場合、構成可能アクセスレジスタです。

CPACR のフォーマットは次のとおりです。

ASEDIS、ビット [31]

アドバンスト SIMD 機能を非稼動にします。

0 このビットによって、命令が未定義になることはありません。

1 *命令のアルファベット順リスト* P. A8-14 ではアドバンスト SIMD の一部として識別され、VFPv3 命令ではない、すべての命令エンコードは未定義です。

実装によって、このビットには次の相違があります。

- VFP が実装され、アドバンスト SIMD が実装されていない場合、このビットは RAO/WI です。
- VFP またはアドバンスト SIMD が実装されていない場合、このビットは UNK/SBZP です。
- VFP とアドバンスト SIMD の両方が実装されている場合、このビットがサポートされるかどうかは実装定義です。サポートされない場合、このビットは RAZ/WI です。

このビットがサポートされている場合は、0 にリセットされます。

D32DIS、ビット [30]

VFP レジスタファイルの D16 ～ D31 の使用を不可能にします。

0 このビットによって、命令が未定義になることはありません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

1 All instruction encodings identified in the *Alphabetical list of instructions* on page A8-14 as being VFPv3 instructions are UNDEFINED if they access any of registers D16-D31.

If this bit is 1 when CPACR.ASEDIS == 0, the result is UNPREDICTABLE.

On an implementation that:

- Does not implement VFP, this bit is UNK/SBZP.
- Implements VFP and does not implement D16-D31, this bit is RAO/WI.
- Implements VFP and implements D16-D31, it is IMPLEMENTATION DEFINED whether this bit is supported. If it is not, then this bit is RAZ/WI.

This bit resets to 0 if it is supported.

Bits [29:28] Reserved. UNK/SBZP.

cp<n>, bits [2n+1, 2n], for n = 0 to 13

Defines the access rights for coprocessor n. The possible values of the field are:

0b00 Access denied. Any attempt to access the coprocessor generates an Undefined Instruction exception.

0b01 Privileged access only. Any attempt to access the coprocessor in User mode generates an Undefined Instruction exception.

0b10 Reserved. The effect of this value is UNPREDICTABLE.

0b11 Full access. The meaning of full access is defined by the appropriate coprocessor.

The value for a coprocessor that is not implemented is 0b00, access denied.

When the Security Extensions are implemented, the NSACR controls whether each coprocessor can be accessed from the Non-secure state, see *c1, Non-Secure Access Control Register (NSACR)* on page B3-110. When the NSACR permits Non-secure access to a coprocessor the level of access permitted is determined by the CPACR. Because the CPACR is not banked, the options for Non-secure state access to a coprocessor are:

- no access
- identical access rights to the Secure state.

If more than one coprocessor is used to provide a set of functionality then having different values for the CPACR fields for those coprocessors can lead to UNPREDICTABLE behavior. An example where this must be considered is with the VFP extension. This uses CP10 and CP11.

Typically, an operating system uses this register to control coprocessor resource sharing among applications:

- Initially all applications are denied access to the shared coprocessor-based resources.
- When an application attempts to use a resource it results in an Undefined Instruction exception.
- The Undefined Instruction handler can then grant access to the resource by setting the appropriate field in the CPACR.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

1 レジスタ D16 ～ D31 のいずれかにアクセスする場合、*命令のアルファベット順リスト* P. A8-14 で VFPv3 命令として識別されるすべての命令エンコードは未定義です。

このビットが 1 で、CPACR.ASEDIS == 0 の場合、結果は予測不能です。

実装によって、このビットには次の相違があります。

- VFP が実装されていない場合、このビットは UNK/SBZP です。
- VFP が実装され、D16 ～ D31 が実装されていない場合、このビットは RAO/WI です。
- VFP と D16 ～ D31 の両方が実装されている場合、このビットがサポートされるかどうかは実装定義です。サポートされない場合、このビットは RAZ/WI です。

このビットがサポートされている場合は、0 にリセットされます。

ビット[29:28] 予約。UNK/SBZP

cp<n>、ビット [2n+1、2n]、n = 0 ～ 13 の場合

コプロセッサ n のアクセス権限を定義します。このフィールドの可能な値は次のとおりです。

0b00 アクセス拒否。コプロセッサにアクセスを試みると、未定義命令例外が発生します。

0b01 特権アクセスのみ。ユーザモードでコプロセッサにアクセスを試みると、未定義命令例外が発生します。

0b10 予約。この値の影響は、予測不能です。

0b11 完全アクセス。完全アクセスの意味は、該当するコプロセッサによって定義されます。

実装されていないコプロセッサの値は、アクセス拒否を表す 0b00 です。

セキュリティ拡張機能が実装されている場合、NSACR によって、非セキュア状態で各コプロセッサにアクセス可能かどうかが制御されます。*c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3-110 を参照して下さい。NSACR でコプロセッサへの非セキュアアクセスが許可されている場合は、許可されるアクセスレベルが CPACR によって決定されます。CPACR はバンクされないため、コプロセッサへの非セキュア状態アクセスの選択肢は次のとおりです。

- アクセス不可
- セキュア状態と同様のアクセス権限

複数のコプロセッサを使用して一連の機能が提供されている場合は、それらのコプロセッサの CPACR フィールドに別々の値を設定すると、予測不能動作につながる可能性があります。この考慮が必要な例は、VFP 拡張機能です。この場合は、CP10 と CP11 が使用されます。

通常、オペレーティングシステムでは、このレジスタを使用して、アプリケーション間で共有されるコプロセッサリソースが制御されます。

- 初期状態では、すべてのアプリケーションが、共有されているコプロセッサベースのリソースへのアクセスを拒否されます。
- アプリケーションがリソースの使用を試みると、未定義命令例外が発生します。
- 未定義命令ハンドラで、CPACR の該当するフィールドを設定することによって、リソースに対するアクセス権限を付与することができます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For details of how this register can be used to check for implemented coprocessors see *Access controls on CP0 to CP13* on page B1-63.

Sharing resources among applications requires a state saving mechanism. Two possibilities are:

- during a context switch, if the last executing process or thread had access rights to a coprocessor then the operating system saves the state of that coprocessor
- on receiving a request for access to a coprocessor, the operating system saves the old state for that coprocessor with the last process or thread that accessed it.

Accessing the CPACR

To access the CPACR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c1, `<CRm>` set to c0, and `<opc2>` set to 2. For example:

```
MRC p15,0,<Rt>,c1,c0,2    ; Read CP15 Coprocessor Access Control Register
MCR p15,0,<Rt>,c1,c0,2    ; Write CP15 Coprocessor Access Control Register
```

Normally, software uses a read, modify, write sequence to update the CPACR, to avoid unwanted changes to the access settings for other coprocessors.

B3.12.20 c1, Secure Configuration Register (SCR)

The Secure Configuration Register, SCR, is part of the Security Extensions.

The SCR defines the configuration of the current security state. It specifies:

- the security state of the processor, Secure or Non-secure
- what mode the processor branches to if an IRQ, FIQ or external abort occurs
- whether the CPSR.F and CPSR.A bits can be modified when SCR.NS = 1.

The SCR:

- is present only when the Security Extensions are implemented
- is a 32-bit read/write register
- is accessible in Secure privileged modes only
- has a defined reset value of 0
- is a Restricted access register, meaning it exists only in the Secure state.

The format of the SCR is:

Bits [31:7] Reserved. UNK/SBZP.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

このレジスタを使用して、実装されているコプロセッサをチェックする方法の詳細については、*CP0 ～ CP13 のアクセス制御* P. B1-63 を参照して下さい。

アプリケーション間でリソースを共有するには、状態保存機構が必要です。次のような 2 種類の方法で実現が可能です。

- コンテキストスイッチ時に、最後に実行されていたプロセスまたはスレッドにコプロセッサへのアクセス権限が付与されている場合は、オペレーティングシステムでそのコプロセッサの状態を保存します。
- コプロセッサへのアクセス要求を受け取ったときに、オペレーティングシステムが、そのコプロセッサの古い状態と、そのコプロセッサにアクセスした最後のプロセスまたはスレッドを保存します。

CPACR へのアクセス

CPACR にアクセスするには、`<opc1>` を 0、`<CRn>` を c1、`<CRm>` を c0、`<opc2>` を 2 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c1,c0,2    ; Read CP15 Coprocessor Access Control Register
MCR p15,0,<Rt>,c1,c0,2    ; Write CP15 Coprocessor Access Control Register
```

通常は、ソフトウェアで読み出し－変更－書き込みシーケンスを使用して CPACR を更新することによって、他のコプロセッサのアクセス設定に対する不要な変更を回避します。

B3.12.20 c1、セキュア構成レジスタ (SCR)

セキュア構成レジスタ (SCR) は、セキュリティ拡張機能の一部です。

SCR によって、現在のセキュリティ状態の構成が定義されます。次のものが指定されます。

- セキュアまたは非セキュアのプロセッサのセキュリティ状態
- IRQ、FIQ、または外部アボートが発生した場合にプロセッサが分岐するモード
- SCR.NS = 1 の場合に CPSR.F ビットと CPSR.A ビットを変更できるかどうか。

SCR の特徴は次のとおりです。

- セキュリティ拡張機能が実装されている場合にのみ存在します。
- 32 ビットの読み出し / 書き込みレジスタ
- セキュア特権モードでのみアクセス可能
- リセット時の値は 0 に定義されています。
- 限定アクセスレジスタで、セキュア状態でのみ存在します。

SCR のフォーマットは次のとおりです。

ビット [31:7] 予約。UNK/SBZP

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

nET, bit [6] Not Early Termination. This bit disables early termination:

0 Early termination permitted. Execution time of data operations can depend on the data values.

1 Disable early termination. The number of cycles required for data operations is forced to be independent of the data values.

This IMPLEMENTATION DEFINED mechanism can be used to disable data dependent timing optimizations from multiplies and data operations. It can provide system support against information leakage that might be exploited by timing correlation types of attack.

On implementations that do not have early termination, this bit is UNK/SBZP.

AW, bit [5] A bit writable. This bit controls whether the A bit in the CPSR can be modified in Non-secure state:

0 the CPSR.A bit can be modified only in Secure state.

1 the CPSR.A bit can be modified in any security state.

For more information, see *Control of aborts by the Security Extensions* on page B1-41.

FW, bit [4] F bit writable. This bit controls whether the F bit in the CPSR can be modified in Non-secure state:

0 the CPSR.F bit can be modified only in Secure state

1 the CPSR.F bit can be modified in any security state.

For more information, see *Control of FIQs by the Security Extensions* on page B1-42.

EA, bit [3] External Abort handler. This bit controls which mode handles external aborts:

0 Abort mode handles external aborts

1 Monitor mode handles external aborts.

For more information, see *Control of aborts by the Security Extensions* on page B1-41.

FIQ, bit [2] FIQ handler. This bit controls which mode the processor enters when a Fast Interrupt (FIQ) is taken:

0 FIQ mode entered when FIQ is taken

1 Monitor mode entered when FIQ is taken.

For more information, see *Control of FIQs by the Security Extensions* on page B1-42.

IRQ, bit [1] IRQ handler. This bit controls which mode the processor enters when an Interrupt (IRQ) is taken:

0 IRQ mode entered when IRQ is taken

1 Monitor mode entered when IRQ is taken.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

nET、ビット[6] 早期終了禁止。このビットによって早期終了が禁止されます。

0 早期終了を許可します。データ操作の実行時間はデータの値に依存する可能性があります。

1 早期終了を禁止します。データ操作に必要なサイクル数は、データの値と無関係に設定されます。

この実装定義の機構は、乗算とデータ処理について、データ依存のタイミング最適化を禁止するために使用できます。また、時差相関型の攻撃に利用される可能性がある、情報漏洩に対するシステムサポートを提供することができます。

早期終了が実装されていない場合、このビットは UNK/SBZP です。

AW、ビット[5] A ビット書き込み可能。このビットによって、非セキュア状態で CPSR の A ビットを変更できるかどうかが制御されます。

0 CPSR.A ビットは、セキュア状態でのみ変更できます。

1 CPSR.A ビットは、任意のセキュリティ状態で変更できます。

詳細については、*セキュリティ拡張機能によるアボートの制御* P. B1-41 を参照して下さい。

FW、ビット[4] F ビット書き込み可能。このビットによって、非セキュア状態で CPSR の F ビットを変更できるかどうかが制御されます。

0 CPSR.F ビットは、セキュア状態でのみ変更できます。

1 CPSR.F ビットは、任意のセキュリティ状態で変更できます。

詳細については、*セキュリティ拡張機能による FIQ の制御* P. B1-42 を参照して下さい。

EA、ビット[3] 外部アボートハンドラ。このビットによって、外部アボートを処理するモードが制御されます。

0 外部アボートをアボートモードで処理します。

1 外部アボートをモニタモードで処理します。

詳細については、*セキュリティ拡張機能によるアボートの制御* P. B1-41 を参照して下さい。

FIQ、ビット[2] FIQ ハンドラ。このビットによって、高速割り込み (FIQ) が取得されたときにプロセッサが遷移するモードが制御されます。

0 FIQ が取得されると、FIQ モードに遷移します。

1 FIQ が取得されると、モニタモードに遷移します。

詳細については、*セキュリティ拡張機能による FIQ の制御* P. B1-42 を参照して下さい。

IRQ、ビット[1] IRQ ハンドラ。このビットによって、割り込み (IRQ) が取得されたときにプロセッサが遷移するモードが制御されます。

0 IRQ が取得されると、IRQ モードに遷移します。

1 IRQ が取得されると、モニタモードに遷移します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

NS, bit [0] Non Secure bit. Except when the processor is in Monitor mode, this bit determines the security state of the processor. Table B3-27 shows the security settings:

Table B3-27 Processor security state

SCR.NS	Processor mode, from CPSR.M bits	
	Monitor mode	All modes except Monitor mode
0	Secure state	Secure state
1	Secure state	Non-secure state

For more information, see *Changing from Secure to Non-secure state* on page B1-27.

The value of the NS bit also affects the accessibility of the Banked CP15 registers in Monitor mode, see *Access to registers in Monitor mode* on page B3-77.

Unless the processor is in Debug state, when an exception occurs in Monitor mode the hardware sets the NS bit to 0.

Whenever the processor changes security state, the monitor code can change the value of the EA, FIQ and IQ bits. This means that the behavior of IRQ, FIQ and External Abort exceptions can be different in each security state.

Accessing the SCR

To access the SCR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c1, `<CRm>` set to c1, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c1,c1,0    ; Read CP15 Secure Configuration Register
MCR p15,0,<Rt>,c1,c1,0    ; Write CP15 Secure Configuration Register
```

B3.12.21 c1, Secure Debug Enable Register (SDER)

The Secure Debug Enable Register, SDER, is part of the Security Extensions.

The SDER controls invasive and non-invasive debug in Secure User mode.

The SDER is:

- present only when the Security Extensions are implemented
- a 32-bit read/write register
- a Restricted access register, meaning it exists only in the Secure state
- accessible in Secure privileged modes only.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

NS、ビット [0] 非セキュアビット。プロセッサがモニタモードの場合を除いて、このビットによって、プロセッサのセキュリティ状態が決定されます。セキュリティ設定を、表 B3-27 に示します。

表 B3-27 プロセッサのセキュリティ状態

SCR.NS	プロセッサモード、CPSR.M ビットから	
	モニタモード	モニタモード以外のすべてのモード
0	セキュア状態	セキュア状態
1	セキュア状態	非セキュア状態

詳細については、*セキュア状態から非セキュア状態への変更* P. B1-27 を参照して下さい。

NS ビットの値は、モニタモードでのバンク CP15 レジスタのアクセス可能性にも影響を与えます。*モニタモードでのレジスタへのアクセス* P. B3-77 を参照して下さい。

プロセッサがデバッグ状態でなければ、モニタモードで例外が発生したときに、ハードウェアで NS ビットが 0 にセットされます。

プロセッサがセキュリティ状態を変更するとき、モニタコードは EA、FIQ、IQ ビットの値を変更できます。これは、IRQ、FIQ、および外部アボート例外の動作が、セキュリティ状態によって異なる可能性があることを意味します。

SCR へのアクセス

SCR にアクセスするには、`<opc1>` を 0、`<CRn>` を c1、`<CRm>` を c1、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c1,c1,0    ; Read CP15 Secure Configuration Register
MCR p15,0,<Rt>,c1,c1,0    ; Write CP15 Secure Configuration Register
```

B3.12.21 c1、セキュアデバッグ イネーブルレジスタ (SDER)

セキュアデバッグ イネーブルレジスタ (SDER) は、セキュリティ拡張機能の一部です。

SDER によって、セキュアユーザ モードでの侵襲性デバッグと非侵襲性デバッグが制御されます。

SDER の特徴は次のとおりです。

- セキュリティ拡張機能が実装されている場合のみ存在します。
- 32 ビットの読み出し / 書き込みレジスタ
- 限定アクセスレジスタで、セキュア状態でのみ存在します。
- セキュア特権モードでのみアクセス可能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the SDER is:

Bits [31:2] Reserved. UNK/SBZP.

SUNIDEN, bit [1]

Secure User Non-Invasive Debug ENable:

0 non-invasive debug not permitted in Secure User mode

1 non-invasive debug permitted in Secure User mode.

SUIDEN, bit [0]

Secure User Invasive Debug ENable:

0 invasive debug not permitted in Secure User mode

1 invasive debug permitted in Secure User mode.

For more information about the use of the SUNIDEN and SUIDEN bits see:

- Chapter C2 *Invasive Debug Authentication*
- Chapter C7 *Non-invasive Debug Authentication*.

Note

Invasive and non-invasive debug in Secure privileged modes is controlled by hardware only. For more information, see Chapter C2 *Invasive Debug Authentication* and Chapter C7 *Non-invasive Debug Authentication*.

Accessing the SDER

To access the SDER you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c1, `<CRm>` set to c1, and `<opc2>` set to 1. For example:

```
MRC p15,0,<Rt>,c1,c1,1    ; Read CP15 Secure Debug Enable Register
MCR p15,0,<Rt>,c1,c1,1    ; Write CP15 Secure Debug Enable Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

SDER のフォーマットは次のとおりです。

ビット [31:2] 予約。UNK/SBZP

SUNIDEN、ビット [1]

セキュアユーザ非侵襲性デバッグイネーブル

0 セキュアユーザ モードでの非侵襲性デバッグは許可されません。

1 セキュアユーザ モードでの非侵襲性デバッグが許可されます。

SUIDEN、ビット [0]

セキュアユーザ侵襲性デバッグイネーブル

0 セキュアユーザ モードでの侵襲性デバッグは許可されません。

1 セキュアユーザ モードでの侵襲性デバッグが許可されます。

SUNIDEN ビットと SUIDEN ビットの使用方法については、次の各章を参照して下さい。

- 第 C2 章*「侵襲性デバッグの認証」*
- 第 C7 章*「非侵襲性デバッグの認証」*

注

セキュア特権モードでの侵襲性デバッグと非侵襲性デバッグは、ハードウェアでのみ制御されます。詳細については、第 C2 章*「侵襲性デバッグの認証」*と第 C7 章*「非侵襲性デバッグの認証」*を参照して下さい。

SDER へのアクセス

SDER にアクセスするには、`<opc1>` を 0、`<CRn>` を c1、`<CRm>` を c1、`<opc2>` を 1 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c1,c1,1    ; Read CP15 Secure Debug Enable Register
MCR p15,0,<Rt>,c1,c1,1    ; Write CP15 Secure Debug Enable Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.22 c1, Non-Secure Access Control Register (NSACR)

The Non-Secure Access Control Register, NSACR, is part of the Security Extensions.

The NSACR defines the Non-secure access permissions to the coprocessors CP0 to CP13. Additional IMPLEMENTATION DEFINED bits in the register can be used to define Non-secure access permissions for IMPLEMENTATION DEFINED functionality.

The NSACR is:

- Present only when the Security Extensions are implemented.
- A 32-bit register
- A Restricted access register. NSACR exists only in the Secure state, but can be read from Non-secure state.
- Accessible only in privileged modes, with access rights that depend on the mode and security state:
  - — the NSACR is read/write in Secure privileged modes
  - — the NSACR is read-only in Non-secure privileged modes.

The format of the NSACR is:

Bits [31:20]

Reserved. UNK/SBZP.

RFR, bit [19] Reserve FIQ Registers:

0 FIQ mode and the FIQ banked registers are accessible in Secure and Non-secure security states.

1 FIQ mode and the FIQ banked registers are accessible in the Secure security state only. Any attempt to access any FIQ Banked register or to enter an FIQ mode when in the Non-secure security states is UNPREDICTABLE.

This bit resets to 0. On some implementations this bit cannot be set to 1.

If NSACR.RFR == 1 when SCR.FIQ == 0, instruction execution is UNPREDICTABLE in Non-secure security state.

Bits [18:16] IMPLEMENTATION DEFINED.

These bits can be used to define the Non-secure access to IMPLEMENTATION DEFINED features.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.22 c1、非セキュアアクセス制御レジスタ (NSACR)

非セキュアアクセス制御レジスタ (NSACR) は、セキュリティ拡張機能の一部です。

NSACR では、コプロセッサ CP0 ～ CP13 に対する非セキュアアクセス許可が定義されます。このレジスタに含まれる追加の実装定義のビットを使用して、実装定義の機能に対する非セキュアアクセス許可を定義できます。

NSACR の特徴は次のとおりです。

- セキュリティ拡張機能が実装されている場合のみ存在します。
- 32 ビットレジスタ
- 限定アクセスレジスタ。NSACR はセキュア状態でのみ存在しますが、非セキュア状態から読み出すことができます。
- 特権モードでのみアクセス可能で、モードとセキュリティ状態によってアクセス権限が異なります。
  - ー セキュア特権モードでは、NSACR は読み出し / 書き込みです。
  - ー 非セキュア特権モードでは、NSACR は読み出し専用です。

NSACR のフォーマットは次のとおりです。

ビット [31:20] 予約。UNK/SBZP

RFR、ビット [19] FIQ レジスタの保存

0 FIQ モードと FIQ バンクレジスタには、セキュアおよび非セキュアのセキュリティ状態でアクセスできます。

1 FIQ モードと FIQ バンクレジスタには、セキュアセキュリティ状態でのみアクセスできます。非セキュアセキュリティ状態で、FIQ バンクレジスタへのアクセス、または FIQ モードへの遷移を試みた場合は予測不能です。

リセット時の値は 0 です。実装によっては、このビットを 1 にセットできない場合があります。

SCR.FIQ == 0 のときに NSACR.RFR == 1 の場合、非セキュアセキュリティ状態での命令実行は予測不能です。

ビット [18:16] 実装定義

これらのビットを使用して、実装定義の機能に対する非セキュアアクセスを定義できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

NSASEDIS, bit[15]

Disable Non-secure Advanced SIMD functionality:

0 This bit has no effect on the ability to write CPACR.ASEDIS.

1 When executing in Non-secure state the CPACR.ASEDIS bit has a fixed value of 1 and writes to it are ignored.

On an implementation that:

- Implements VFP and does not implement Advanced SIMD, this bit is RAO/WI.
- Does not implement VFP or Advanced SIMD, this bit is UNK/SBZP.
- Implements both VFP and Advanced SIMD, it is IMPLEMENTATION DEFINED whether this bit is supported. If it is not supported it is RAZ/WI.

This bit resets to 0 if it is supported.

NSD32DIS, bit[14]

Disable Non-secure use of D16-D31 of the VFP register file:

0 This bit has no effect on the ability to write CPACR.D32DIS.

1 When executing in Non-secure state, the CPACR.D32DIS bit has a fixed value of 1 and writes to it are ignored.

If this bit is 1 when NSACR.NSASEDIS == 0, the result is UNPREDICTABLE.

On an implementation that:

- Does not implement VFP, this bit is UNK/SBZP.
- Implements VFP and does not implement D16-D31, this bit is RAO/WI.
- Implements VFP and implements D16-D31, it is IMPLEMENTATION DEFINED whether this bit is supported. If it is not supported it is RAZ/WI.

This bit resets to 0 if it is supported.

cp<n>, bit [n], for n = 0 to 13

Non-secure access to coprocessor <n> enable. Each bit enables access to the corresponding coprocessor from Non-secure state:

0 Coprocessor <n> can be accessed only from Secure state. Any attempt to access coprocessor <n> in Non-secure state results in an Undefined Instruction exception.

If the processor is in Non-secure state it cannot write the corresponding bits in the CPACR, and reads them as 0b00, access denied.

1 Coprocessor <n> can be accessed from any security state.

If Non-secure access to a coprocessor is enabled, the CPACR must be checked to determine the level of access that is permitted, see *c1, Coprocessor Access Control Register (CPACR)* on page B3-104.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

NSASEDIS、ビット [15]

非セキュアアドバンスト SIMD 機能を不可能にします。

0 このビットは、CPACR.ASEDIS への書き込みに影響を与えません。

1 非セキュア状態で実行している場合は、CPACR.ASEDIS ビットの値が 1 に固定され、書き込みは無視されます。

実装によって、このビットには次の相違があります。

- VFP が実装され、アドバンスト SIMD が実装されていない場合、このビットは RAO/WI です。
- VFP またはアドバンスト SIMD が実装されていない場合、このビットは UNK/SBZP です。
- VFP とアドバンスト SIMD の両方が実装されている場合、このビットがサポートされるかどうかは実装定義です。サポートされない場合、このビットは RAZ/WI です。

このビットがサポートされている場合は、0 にリセットされます。

NSD32DIS、ビット [14]

VFP レジスタファイル D16 ～ D31 の非セキュア使用を不可能にします。

0 このビットは、CPACR.D32DIS への書き込みに影響を与えません。

1 非セキュア状態で実行している場合は、CPACR.D32DIS ビットの値が 1 に固定され、書き込みは無視されます。

このビットが 1 で、NSACR.NSASEDIS == 0 の場合、結果は予測不能です。

実装によって、このビットには次の相違があります。

- VFP が実装されていない場合、このビットは UNK/SBZP です。
- VFP が実装され、D16 ～ D31 が実装されていない場合、このビットは RAO/WI です。
- VFP と D16 ～ D31 の両方が実装されている場合、このビットがサポートされるかどうかは実装定義です。サポートされない場合、このビットは RAZ/WI です。

このビットがサポートされている場合は、0 にリセットされます。

cp<n>、ビット [n]、n = 0 ～ 13 の場合

コプロセッサ <n> への非セキュアアクセス可能。各ビットは、対応するコプロセッサへの非セキュア状態からのアクセスを可能にします。

0 コプロセッサ <n> には、セキュア状態からのみアクセスできます。非セキュア状態でコプロセッサ <n> にアクセスを試みると、未定義命令例外が発生します。

プロセッサが非セキュア状態の場合は、CPACR の対応するビットに書き込むことができません。このレジスタを読み出した場合、アクセス拒否を表す 0b00 が返されます。

1 コプロセッサ <n> には、任意のセキュリティ状態からアクセスできます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If multiple coprocessors are used to control a feature then the Non-secure access enable bits for those coprocessors must be set to the same value, otherwise behavior is UNPREDICTABLE. For example, when the VFP extension is implemented it is controlled by coprocessors 10 and 11, and bits [10,11] of the NSACR must be set to the same value.

For bits that correspond to coprocessors that are not implemented, it is IMPLEMENTATION DEFINED whether the bits:

- behave as RAZ/WI
- can be written by Secure privileged modes.

Accessing the NSACR

To access the NSACR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c1, `<CRm>` set to c1, and `<opc2>` set to 2. For example:

```
MRC p15,0,<Rt>,c1,c1,2    ; Read CP15 Non-Secure Access Control Register
MCR p15,0,<Rt>,c1,c1,2    ; Write CP15 Non-Secure Access Control Register
```

You can write to the NSACR only in Secure privileged modes.

You can read the register in any privileged mode.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

コプロセッサへの非セキュアアクセスが可能な場合は、CPACR をチェックして、許可されているアクセスレベルを判定する必要があります。*c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B3-104 を参照して下さい。

複数のコプロセッサを使用して 1 つの機能が制御される場合は、それらのコプロセッサの非セキュアアクセス イネーブルビットを同じ値にセットする必要があります。そうでない場合、動作は予測不能です。例えば、VFP 拡張機能が実装されている場合、コプロセッサ 10 および 11 によって制御されるため、NSACR のビット [10、11] を同じ値にセットする必要があります。

実装されていないコプロセッサに対応するビットについて、次のどちらを採用するかは実装定義です。

- RAZ/WI として動作する。
- セキュア特権モードで書き込みできる。

NSACR へのアクセス

NSACR にアクセスするには、`<opc1>` を 0、`<CRn>` を c1、`<CRm>` を c1、`<opc2>` を 2 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c1,c1,2    ; Read CP15 Non-Secure Access Control Register
MCR p15,0,<Rt>,c1,c1,2    ; Write CP15 Non-Secure Access Control Register
```

NSACR には、セキュア特権モードでのみ書き込みできます。

このレジスタは、任意の特権モードで読み出しできます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.23 CP15 c2 and c3, Memory protection and control registers

On an ARMv7-A implementation, the CP15 c2 and c3 registers are used for memory protection and control. Figure B3-14 shows these registers.

Figure B3-14 CP15 c2 and c3 registers

CP15 c2 and c3 register encodings not shown in Figure B3-14 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B3-69.

B3.12.24 CP15 c2, Translation table support registers

When the VMSA is implemented, three translation table support registers are implemented in CP15 c2. Table B3-28 summarizes these registers.

Table B3-28 VMSA translation table support registers

Register name	Description
Translation Table Base 0	*c2, Translation Table Base Register 0 (TTBR0)*
Translation Table Base 1	*c2, Translation Table Base Register 1 (TTBR1)* on page B3-116
Translation Table Base Control	*c2, Translation Table Base Control Register (TTBCR)* on page B3-117

The description of the TTBCR describes the use of this set of registers, see *c2, Translation Table Base Control Register (TTBCR)* on page B3-117.

c2, Translation Table Base Register 0 (TTBR0)

The Translation Table Base Register 0, TTBR0, holds the base address of translation table 0, and information about the memory it occupies.

The TTBR0 register:

- is a 32-bit read/write register
- is accessible only in privileged modes
- when the Security Extensions are implemented:
  - — is a Banked register.
  - — has write access to the Secure copy of the register disabled when the **CP15SDISABLE** signal is asserted HIGH.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.23 CP15 c2 および c3、メモリ保護および制御レジスタ

ARMv7-A 実装では、CP15 c2 および c3 レジスタがメモリ保護および制御に使用されます。これらのレジスタを、図 B3-14 に示します。

図 B3- 14 CP15 c2 および c3 レジスタ

図 B3- 14 に示されていない CP15 c2 および c3 レジスタのエンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B3- 69 を参照して下さい。

B3.12.24 CP15 c2、変換テーブルサポート レジスタ

VMSA が実装されている場合は、3 つの変換テーブルサポート レジスタが CP15 c2 に実装されます。これらのレジスタの概要を、表 B3- 28 に示します。

表 B3- 28 VMSA 変換テーブルサポート レジスタ

レジスタ名	説明
変換テーブルベース 0	*c2、変換テーブルベースレジスタ 0 (TTBR0)*
変換テーブルベース 1	*c2、変換テーブルベースレジスタ 1(TTBR1)* P. B3- 116
変換テーブルベース制御	*c2、変換テーブルベース制御レジスタ (TTBCR)* P. B3- 117

TTBCR の説明で、このレジスタセットの使用方法を解説します。*c2、変換テーブルベース制御レジスタ (TTBCR)* P. B3- 117 を参照して下さい。

c2、変換テーブルベースレジスタ 0 (TTBR0)

変換テーブルベースレジスタ 0 (TTBR0) には、変換テーブル 0 のベースアドレスと、それによって占有されるメモリについての情報が保持されます。

TTBR0 レジスタの特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、次の特徴があります。
  - – バンクレジスタ
  - – レジスタのセキュアコピーへの書き込みアクセスは、CP15SDISABLE 信号が HIGH にアサートされているときには不可能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When the Multiprocessing Extensions are not implemented, the format of the TTBR0 register is:

When the Multiprocessing Extensions are implemented, the format of the TTBR0 register is:

Bits [31:14-N] Translation table base 0 address, bits [31:14-N].

The value of N determines the required alignment of the translation table, which must be aligned to 2^{14-N} bytes.

Bits [13-N:6], ARMv7-A base architecture

UNK/SBZP.

Bits [13-N:7], when the Multiprocessing Extensions are implemented

UNK/SBZP.

IRGN[0], bit [6], when the Multiprocessing Extensions are implemented

See the description of bit [0] when the Multiprocessing Extensions are implemented.

NOS, bit [5] Not Outer Shareable bit. Indicates the Outer Shareable attribute for the memory associated with a translation table walk that has the Shareable attribute, indicated by TTBR0.S == 1:

0 Outer Shareable

1 Inner Shareable.

This bit is ignored when TTBR0.S == 0.

This bit is only implemented from ARMv7.

RGN, bits [4:3]

Region bits. Indicates the Outer Cacheability attributes for the memory associated with the translation table walks:

0b00 Normal memory, Outer Non-cacheable

0b01 Normal memory, Outer Write-Back Write-Allocate Cacheable

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

マルチプロセッシング拡張機能が実装されていない場合、TTBR0 レジスタのフォーマットは次のとおりです。

マルチプロセッシング拡張機能が実装されている場合、TTBR0 レジスタのフォーマットは次のとおりです。

ビット[31:14-N] 変換テーブルベース 0 のアドレス、ビット [31:14-N]。

N の値によって、2^{14-N} バイトにアラインする必要のある変換テーブルのアライメントが決定されます。

ビット [13-N:6]、ARMv7-A 基本アーキテクチャ

UNK/SBZP

ビット [13-N:7]、マルチプロセッシング拡張機能が実装されている場合

UNK/SBZP

IRGN[0]、ビット [6]、マルチプロセッシング拡張機能が実装されている場合

マルチプロセッシング拡張機能が実装されている場合の、ビット [0] の説明を参照して下さい。

NOS、ビット[5] 非外部共有可能ビット。変換テーブルウォークについて、TTBR0.S == 1 で共有可能属性が示されている場合、その変換テーブルウォークに関連付けられたメモリの外部共有可能属性を示します。

0 外部共有可能

1 内部共有可能

TTBR0.S == 0 の場合、このビットは無視されます。

このビットは、ARMv7 およびそれ以降にのみ実装されています。

RGN、ビット [4:3]

領域ビット。変換テーブルウォークに関連付けられたメモリの、外部キャッシュ可能属性を示します。

0b00 ノーマルメモリ、外部キャッシュ不可

0b01 ノーマルメモリ、外部ライトバック書き込み割り当てキャッシュ可能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b10 Normal memory, Outer Write-Through Cacheable

0b11 Normal memory, Outer Write-Back no Write-Allocate Cacheable.

IMP, bit [2] The effect of this bit is IMPLEMENTATION DEFINED. If the translation table implementation does not include any IMPLEMENTATION DEFINED features this bit is SBZ.

S, bit [1] Shareable bit. Indicates the Shareable attribute for the memory associated with the translation table walks:

0 Non-shareable

1 Shareable.

C, bit [0], ARMv7-A base architecture

Cacheable bit. Indicates whether the translation table walk is to Inner Cacheable memory.

0 Inner Non-cacheable

1 Inner Cacheable.

For regions marked as Inner Cacheable, it is IMPLEMENTATION DEFINED whether the read has the Write-Through, Write-Back no Write-Allocate, or Write-Back Write-Allocate attribute.

IRGN, bits [6,0], when the Multiprocessing Extensions are implemented

Inner region bits. Indicates the Inner Cacheability attributes for the memory associated with the translation table walks. The possible values of IRGN[1:0] are:

0b00 Normal memory, Inner Non-cacheable

0b01 Normal memory, Inner Write-Back Write-Allocate Cacheable

0b10 Normal memory, Inner Write-Through Cacheable

0b11 Normal memory, Inner Write-Back no Write-Allocate Cacheable.

Note

The encoding of the IRGN bits is counter-intuitive, with register bit [6] being IRGN[0] and register bit [0] being IRGN[1]. This encoding is chosen to give a consistent encoding of memory region types and to ensure that software written for the ARMv7 base architecture can run unmodified on an implementation that includes the Multiprocessing Extensions.

Accessing the TTBR0 register

To access the TTBR0 register you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c2, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c2,c0,0    ; Read CP15 Translation Table Base Register 0
MCR p15,0,<Rt>,c2,c0,0    ; Write CP15 Translation Table Base Register 0
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b10 ノーマルメモリ、外部ライトスルー キャッシュ可能

0b11 ノーマルメモリ、外部ライトバック書き込み割り当てなしキャッシュ可能

IMP、ビット[2] このビットの作用は、実装定義です。変換テーブル実装に実装定義の機能が含まれていない場合、このビットは SBZ です。

S、ビット[1] 共有可能ビット。変換テーブルウォークに関連付けられたメモリの共有可能属性を示します。

0 共有不可

1 共有可能

C、ビット [0]、ARMv7-A 基本アーキテクチャ

キャッシュ可能ビット。変換テーブルウォークが内部キャッシュ可能メモリに対するものかどうかを示します。

0 内部キャッシュ不可

1 内部キャッシュ可能

内部キャッシュ可能としてマークされた領域の場合、読み出しにライトスルー、ライトバック書き込み割り当てなし、またはライトバック書き込み割り当ての属性が付与されるかどうかは実装定義です。

IRGN、ビット [6、0]、マルチプロセッシング拡張機能が実装されている場合

内部領域ビット。変換テーブルウォークに関連付けられたメモリの内部キャッシュ可能属性を示します。IRGN[1:0] の可能な値は次のとおりです。

0b00 ノーマルメモリ、内部キャッシュ不可

0b01 ノーマルメモリ、内部ライトバック書き込み割り当てキャッシュ可能

0b10 ノーマルメモリ、内部ライトスルー キャッシュ可能

0b11 ノーマルメモリ、内部ライトバック書き込み割り当てなしキャッシュ可能

注

IRGN ビットのエンコードは、直感に反してレジスタのビット [6] が IRGN[0]、レジスタのビット [0] が IRGN[1] です。このエンコードは、メモリ領域タイプの一貫したエンコードを提供し、マルチプロセッシング拡張機能を含む実装上で、ARMv7 基本アーキテクチャ用に作成されたソフトウェアを修正せずに実行できることを保証するために選択されたものです。

TTBR0 レジスタへのアクセス

TTBR0 レジスタにアクセスするには、`<opc1>` を 0、`<CRn>` を c2、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c2,c0,0    ; Read CP15 Translation Table Base Register 0
MCR p15,0,<Rt>,c2,c0,0    ; Write CP15 Translation Table Base Register 0
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c2, Translation Table Base Register 1 (TTBR1)

The Translation Table Base Register 1, TTBR1, holds the base address of translation table 1, and information about the memory it occupies.

The TTBR1 register is:

- a 32-bit read/write register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Banked register.

When the Multiprocessing Extensions are not implemented, the format of the TTBR1 register is:

When the Multiprocessing Extensions are implemented, the format of the TTBR1 register is:

Bits [31:14] Translation table base 1 address, bits [31:14]. The translation table must be aligned on a 16KByte boundary.

Bits [13:6], ARMv7-A base architecture

UNK/SBZP.

Bits [13:7], when the Multiprocessing Extensions are implemented

UNK/SBZP.

IRGN[0:1], bits [6,0], when the Multiprocessing Extensions are implemented

See the definition given for the TTBR0 in *c2, Translation Table Base Register 0 (TTBR0)* on page B3-113.

NOS, RGN, IMP, S, bits [5:1]

See the definitions given for the TTBR0 in *c2, Translation Table Base Register 0 (TTBR0)* on page B3-113.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c2、変換テーブルベースレジスタ 1(TTBR1)

変換テーブルベースレジスタ 1(TTBR1) には、変換テーブル 1 のベースアドレスと、それによって占有されるメモリについての情報が保持されます。

TTBR1 レジスタの特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、バンクレジスタです。

マルチプロセッシング拡張機能が実装されていない場合、TTBR1 レジスタのフォーマットは次のとおりです。

マルチプロセッシング拡張機能が実装されている場合、TTBR1 レジスタのフォーマットは次のとおりです。

ビット[31:14] 変換テーブルベース 1 のアドレス、ビット [31:14]。変換テーブルは 16KB 境界にアラインしている必要があります。

ビット[13:6]、ARMv7-A 基本アーキテクチャ

UNK/SBZP

ビット[13:7]、マルチプロセッシング拡張機能が実装されている場合

UNK/SBZP

IRGN[0:1]、ビット [6、0]、マルチプロセッシング拡張機能が実装されている場合

c2、変換テーブルベースレジスタ 0 (TTBR0) P. B3- 113 にある、TTBR0 の定義を参照して下さい。

NOS、RGN、IMP、S、ビット [5:1]

c2、変換テーブルベースレジスタ 0 (TTBR0) P. B3- 113 にある、TTBR0 の定義を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C, bit [0], ARMv7-A base architecture

See the definition given for the TTBR0 in *c2, Translation Table Base Register 0 (TTBR0)* on page B3-113.

Accessing the TTBR1 register

To access the TTBR1 register you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c2, `<CRm>` set to c0, and `<opc2>` set to 1. For example:

```
MRC p15,0,<Rt>,c2,c0,1    ; Read CP15 Translation Table Base Register 1
MCR p15,0,<Rt>,c2,c0,1    ; Write CP15 Translation Table Base Register 1
```

c2, Translation Table Base Control Register (TTBCR)

The Translation Table Base Control Register, TTBCR, determines which of the Translation Table Base Registers, TTBR0 or TTBR1, defines the base address for the translation table walk that is required when an MVA is not found in the TLB.

The TTBCR:

- Is a 32-bit read/write register.
- Is accessible only in privileged modes
- Has a defined reset value of 0. When the Security Extensions are implemented, this reset value applies only to the Secure copy of the register, and software must program the Non-secure copy of the register with the required value.
- When the Security Extensions are implemented:
  - — is a Banked register.
  - — has write access to the Secure copy of the register disabled when the **CP15SDISABLE** signal is asserted HIGH.

When the Security Extensions are not implemented, the format of the TTBCR is:

When the Security Extensions are implemented, the format of the TTBCR is:

Bits [31:6, 3] UNK/SBZP.

PD1, bit [5], when Security Extensions are implemented

Translation table walk Disable bit for TTBR1. This bit controls whether a translation table walk is performed on a TLB miss when TTBR1 is used:

0 If a TLB miss occurs when TTBR1 is used a translation table walk is performed.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C、ビット [0]、ARMv7-A 基本アーキテクチャ

c2、変換テーブルベースレジスタ 0 (TTBR0) P. B3-113 にある、TTBR0 の定義を参照して下さい。

TTBR1 レジスタへのアクセス

TTBR1 レジスタにアクセスするには、`<opc1>` を 0、`<CRn>` を c2、`<CRm>` を c0、`<opc2>` を 1 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c2,c0,1    ; Read CP15 Translation Table Base Register 1
MCR p15,0,<Rt>,c2,c0,1    ; Write CP15 Translation Table Base Register 1
```

c2、変換テーブルベース制御レジスタ (TTBCR)

変換テーブルベース制御レジスタ (TTBCR) は、TLB に MVA が見つからない場合に必要となる変換テーブルウォークのベースアドレスが、変換テーブルベース レジスタの TTBR0 と TTBR1 のどちらで定義されるかを決定します。

TTBCR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- リセット時の値は 0 に定義されています。セキュリティ拡張機能が実装されている場合、このリセット時の値はレジスタのセキュアコピーにのみ適用されるため、ソフトウェアでレジスタの非セキュアコピーを必要な値にプログラムする必要があります。
- セキュリティ拡張機能が実装されている場合、次の特徴があります。
  - – バンクレジスタ
  - – レジスタのセキュアコピーへの書き込みアクセスは、CP15SDISABLE 信号が HIGH にアサートされたときに禁止される

セキュリティ拡張機能が実装されていない場合、TTBCR のフォーマットは次のとおりです。

セキュリティ拡張機能が実装されている場合、TTBCR のフォーマットは次のとおりです。

ビット[31:6、3] UNK/SBZP

PD1、ビット [5]、セキュリティ拡張機能が実装されている場合

TTBR1 用の変換テーブルウォーク ディセーブルビット。このビットによって、TTBR1 使用時の TLB ミスで、変換テーブルウォークが実行されるかどうかが制御されます。

0 TTBR1 の使用時に TLB ミスが発生した場合、変換テーブルウォークが実行されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

1 If a TLB miss occurs when TTBR1 is used no translation table walk is performed and a Section Translation fault is returned.

PD0, bit [4], when Security Extensions are implemented

Translation table walk Disable bit for TTBR0. This bit controls whether a translation table walk is performed on a TLB miss when TTBR0 is used. The meanings of the possible values of this bit are equivalent to those for the PD1 bit.

Bits [5:4], when Security Extensions are not implemented

UNK/SBZP.

N, bits [2:0] Indicate the width of the base address held in TTBR0. In TTBR0, the base address field is bits [31:14-N]. The value of N also determines:

- whether TTBR0 or TTBR1 is used as the base address for translation table walks.
- the size of the translation table pointed to by TTBR0.

N can take any value from 0 to 7, that is, from 0b000 to 0b111.

When N has its reset value of 0, the translation table base is compatible with ARMv5 and ARMv6.

Determining which TTBR to use, and the TTBR0 translation table size

When an MVA is not found in the TLB, the value of TTBCR.N determines whether TTBR0 or TTBR1 is used as the base address for the translation table walk in memory:

- if N == 0 then always use TTBR0
- if N > 0 then:
  - — if bits [31:32-N] of the MVA are all zero then use TTBR0
  - — otherwise use TTBR1.

The size of the first-level translation tables accessed by TTBR0 depends on the value of TTBCR.N as shown in Table B3-29:

Table B3-29 Value of N field and the size of the TTBR0 translation table

TTBCR.N	Size of TTBR0 translation table
0b000	16KB
0b001	8KB
0b010	4KB
0b011	2KB
0b100	1KB

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

1 TTBR1 の使用時に TLB ミスが発生しても、変換テーブルウォークは実行されず、セクション変換フォールトが返されます。

PD0、ビット [4]、セキュリティ拡張機能が実装されている場合

TTBR0 用の変換テーブルウォーク ディセーブルビット。このビットによって、TTBR0 使用時の TLB ミスで、変換テーブルウォークが実行されるかどうかが制御されます。このビットの可能な値の意味は、PD1 ビットと等価です。

ビット [5:4]、セキュリティ拡張機能が実装されていない場合

UNK/SBZP

N、ビット [2:0] TTBR0 に保持されるベースアドレスの幅を示します。TTBR0 では、ベースアドレスフィールドがビット [31:14-N] です。N の値によって、次のものも決定されます。

- 変換テーブルウォークのベースアドレスとして、TTBR0 と TTBR1 のどちらが使用されるか。
- TTBR0 で示される変換テーブルのサイズ

N は、0 ～ 7、つまり 0b000 ～ 0b111 の任意の値です。

N のリセット時の値が 0 の場合は、変換テーブルベースの ARMv5 および ARMv6 との互換性が維持されます。

使用される TTBR と TTBR0 変換テーブルサイズの判定

TLB に MVA が見つからない場合は、TTBCR.N の値によって、TTBR0 と TTBR1 のどちらがメモリ内の変換テーブルウォーク用のベースレジスタとして使用されるかが決定されます。

- N == 0 の場合は、常に TTBR0 が使用されます。
- N > 0 の場合、次のように決定されます。
  - – MVA のビット [31:32-N] がすべて 0 の場合、TTBR0 が使用されます。
  - – それ以外の場合、TTBR1 が使用されます。

表 B3-29 に示すように、TTBR0 によってアクセスされる第 1 レベル変換テーブルのサイズは、TTBCR.N の値によって異なります。

表 B3-29 N フィールドの値と TTBR0 変換テーブルのサイズ

TTBCR.N	TTBR0 変換テーブルのサイズ
0b000	16KB
0b001	8KB
0b010	4KB
0b011	2KB
0b100	1KB

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-29 Value of N field and the size of the TTBR0 translation table (continued)

TTBCR.N	Size of TTBR0 translation table
0b101	512 bytes
0b110	256 bytes
0b111	128 bytes

Accessing the TTBCR

To access the TTBCR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c2, `<CRm>` set to c0, and `<opc2>` set to 2. For example:

```
MRC p15,0,<Rt>,c2,c0,2    ; Read CP15 Translation Table Base Control Register
MCR p15,0,<Rt>,c2,c0,2    ; Write CP15 Translation Table Base Control Register
```

B3.12.25 c3, Domain Access Control Register (DACR)

The Domain Access Control Register, DACR, defines the access permission for each of the sixteen memory domains.

The DACR:

- is a 32-bit read/write register
- is accessible only in privileged modes
- when the Security Extensions are implemented:
  - — is a Banked register.
  - — has write access to the Secure copy of the register disabled when the **CP15SDISABLE** signal is asserted HIGH.

The format of the DACR is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D15		D14		D13		D12		D11		D10		D9		D8		D7		D6		D5		D4		D3		D2		D1		D0	

Dn, bits [(2n+1):2n]

Domain n access permission, where n = 0 to 15. Permitted values are:

0b00 No access. Any access to the domain generates a Domain fault.

0b01 Client. Accesses are checked against the permission bits in the translation tables.

0b10 Reserved, effect is UNPREDICTABLE

0b11 Manager. Accesses are not checked against the permission bits in the translation tables.

For more information, see *Domains* on page B3-31.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B3-29 N フィールドの値と TTBR0 変換テーブルのサイズ（続き）

TTBCR.N	TTBR0 変換テーブルのサイズ
0b101	512 バイト
0b110	256 バイト
0b111	128 バイト

TTBCR へのアクセス

TTBCR にアクセスするには、`<opc1>` を 0、`<CRn>` を c2、`<CRm>` を c0、`<opc2>` を 2 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c2,c0,2    ; Read CP15 Translation Table Base Control Register
MCR p15,0,<Rt>,c2,c0,2    ; Write CP15 Translation Table Base Control Register
```

B3.12.25 c3、ドメインアクセス制御レジスタ (DACR)

ドメインアクセス制御レジスタ (DACR) は、16 個のメモリドメインのそれぞれについて、アクセス許可を定義します。

DACR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、次の特徴があります。
  - – バンクレジスタ
  - – レジスタのセキュアコピーへの書き込みアクセスは、CP15SDISABLE 信号が HIGH にアサートされているときには不可能です。

DACR のフォーマットは次のとおりです。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D15		D14		D13		D12		D11		D10		D9		D8		D7		D6		D5		D4		D3		D2		D1		D0	

Dn、ビット [(2n+1):2n]

ドメイン n のアクセス許可 (n = 0 ～ 15)。許可される値は次のとおりです。

0b00 アクセス不可。ドメインへの任意のアクセスでドメインフォールトが発生します。

0b01 クライアント。アクセスは、変換テーブルのアクセス許可ビットに対してチェックされます。

0b10 予約。結果は予測不能です。

0b11 マネージャ。アクセスは、変換テーブルのアクセス許可ビットに対してチェックされません。

詳細については、*ドメイン* P. B3-31 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the DACR

To access the DACR you read or write the CP15 registers with <opc1> set to 0, <CRn> set to c3, <CRm> set to c0, and <opc2> set to 0. For example:

```
MRC p15,0,<Rt>,c3,c0,0    ; Read CP15 Domain Access Control Register
MCR p15,0,<Rt>,c3,c0,0    ; Write CP15 Domain Access Control Register
```

B3.12.26 CP15 c4, Not used

CP15 c4 is not used on any ARMv7 implementation, see *Unallocated CP15 encodings* on page B3-69.

B3.12.27 CP15 c5 and c6, Memory system fault registers

The CP15 c5 and c6 registers are used for memory system fault reporting. Figure B3-15 shows the CP15 c5 and c6 registers.

Figure B3-15 CP15 c5 and c6 registers in a VMSA implementation

CP15 c5 and c6 register encodings not shown in Figure B3-15 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B3-69.

The CP15 c5 and c6 registers are described in:

- *CP15 c5, Fault status registers* on page B3-121
- *CP15 c6, Fault Address registers* on page B3-124.

Also, these registers are used to report information about debug exceptions. For details see *Effects of debug exceptions on CP15 registers and the DBGWFAR* on page C4-4.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DACR へのアクセス

DACR にアクセスするには、<opc1> を 0、<CRn> を c3、<CRm> を c0、<opc2> を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c3,c0,0    ; Read CP15 Domain Access Control Register
MCR p15,0,<Rt>,c3,c0,0    ; Write CP15 Domain Access Control Register
```

B3.12.26 CP15 c4、未使用

ARMv7 実装では CP15 c4 が使用されません。*未割り当ての CP15 エンコード* P. B3-69 を参照して下さい。

B3.12.27 CP15 c5 および c6、メモリシステム フォールトレジスタ

CP15 c5 および c6 レジスタは、メモリシステム フォールトの報告に使用されます。CP15 c5 および c6 レジスタを、図 B3-15 に示します。

図 B3-15 VMSA 実装の CP15 c5 および c6 レジスタ

図 B3-15 に示されていない CP15 c5 および c6 レジスタのエンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B3-69 を参照して下さい。

CP15 c5 および c6 レジスタについては、次のセクションで説明します。

- *CP15 c5、フォールトステータス レジスタ* P. B3-121
- *CP15 c6、フォールトアドレス レジスタ* P. B3-124

これらのレジスタは、デバッグ例外に関する情報の報告にも使用されます。詳細については、*CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響* P. C4-4 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.28 CP15 c5, Fault status registers

There are two fault status registers, in CP15 c5, and the architecture provides encodings for two additional IMPLEMENTATION DEFINED registers. Table B3-30 summarizes these registers.

Table B3-30 Fault status registers

Register name	Description
Data Fault Status Register (DFSR)	*c5, Data Fault Status Register (DFSR)*
Instruction Fault Status Register (IFSR)	*c5, Instruction Fault Status Register (IFSR)* on page B3-122
Auxiliary Data Fault Status Register (ADFSR)	*c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B3-123
Auxiliary Instruction Fault Status Register (AIFSR)	

Fault information is returned using the fault status registers and the fault address registers described in *CP15 c6, Fault Address registers* on page B3-124. For details of how these registers are used see *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48.

c5, Data Fault Status Register (DFSR)

The Data Fault Status Register, DFSR, holds status information about the last data fault.

The DFSR is:

- a 32-bit read/write register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Banked register.

The format of the DFSR is:

Bits [31:13,9:8]

UNK/SBZP.

ExT, bit [12] External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.28 CP15 c5、フォールトステータス レジスタ

CP15 c5 には 2 つのフォールト ステータス レジスタがあり、アーキテクチャでは、その他に 2 つの実装定義レジスタのエンコードが提供されます。これらのレジスタを、表 B3-30 に示します。

表 B3-30 フォールトステータス レジスタ

レジスタ名	説明
データフォールト ステータスレジスタ (DFSR)	*c5、データフォールト ステータスレジスタ (DFSR)*
命令フォールト ステータスレジスタ (IFSR)	*c5、命令フォールト ステータスレジスタ (IFSR)* P. B3-122
補助データフォールト ステータスレジスタ (ADFSR)	*c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B3-123
補助命令フォールト ステータスレジスタ (AIFSR)	

フォールト情報は、フォールト ステータス レジスタと、*CP15 c6、フォールトアドレス レジスタ* P. B3-124 で説明されているフォールトアドレス レジスタを使用して返されます。これらのレジスタの使用方法の詳細については、*VMSA 実装のフォールト ステータス レジスタとフォールトアドレス レジスタ* P. B3-48 を参照して下さい。

c5、データフォールト ステータスレジスタ (DFSR)

データフォールト ステータスレジスタ (DFSR) には、最後のデータフォールトに関するステータス情報が保持されます。

DFSR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、バンクレジスタです。

DFSR のフォーマットは次のとおりです。

ビット [31:13、9:8]

UNK/SBZP

ExT、ビット[12] 外部アボートタイプ。このビットは、外部アボートの実装定義の分類を提供するために使用できます。

外部アボート以外のアボートでは、このビットは常に 0 を返します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

WnR, bit [11] Write not Read bit. Indicates whether the abort was caused by a write or a read access:

0 Abort caused by a read access

1 Abort caused by a write access.

For faults on CP15 cache maintenance operations, including the VA to PA translation operations, this bit always returns a value of 1.

FS, bits [10,3:0]

Fault status bits. For the valid encodings of these bits in an ARMv7-A implementation with a VMSA, see Table B3-12 on page B3-51.

All encodings not shown in the table are reserved.

Domain, bits [7:4]

The domain of the fault address.

From ARMv7 use of this field is deprecated, see *The Domain field in the DFSR* on page B3-52.

For information about using the DFSR see *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48.

Accessing the DFSR

To access the DFSR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c5, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c5,c0,0    ; Read CP15 Data Fault Status Register
MCR p15,0,<Rt>,c5,c0,0    ; Write CP15 Data Fault Status Register
```

c5, Instruction Fault Status Register (IFSR)

The Instruction Fault Status Register, IFSR, holds status information about the last instruction fault.

The IFSR is:

- a 32-bit read/write register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Banked register.

The format of the IFSR is:

Bits [31:13,11,9:4]

UNK/SBZP.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

WnR、ビット[11] 書き込み / 読み出しビット。アボートが書き込みアクセスと読み出しアクセスのどちらで引き起こされたのかを示します。

0 読み出しアクセスで引き起こされたアボートです。

1 書き込みアクセスで引き起こされたアボートです。

VA から PA への変換操作など、CP15 キャッシュ保守操作でフォールトが発生した場合、このビットは常に 1 の値を返します。

FS、ビット [10、3:0]

フォールトステータス ビット。VMSA を含む ARMv7-A 実装における、これらのビットの有効なエンコードについては、P. B3-51 の表 B3-12 を参照して下さい。

この表に示されていないエンコードはすべて予約されています。

ドメイン、ビット [7:4]

フォールトアドレスのドメイン

ARMv7 およびそれ以降では、このフィールドの使用は非推奨です。*DFSR のドメインフィールド* P. B3-52 を参照して下さい。

DFSR の使用方法については、*VMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B3-48 を参照して下さい。

DFSR へのアクセス

DFSR にアクセスするには、`<opc1>` を 0、`<CRn>` を c5、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c5,c0,0    ; Read CP15 Data Fault Status Register
MCR p15,0,<Rt>,c5,c0,0    ; Write CP15 Data Fault Status Register
```

c5、命令フォールト ステータスレジスタ (IFSR)

命令フォールト ステータスレジスタ (IFSR) には、最後の命令フォールトに関するステータス情報が保持されます。

IFSR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、バンクレジスタです。

IFSR のフォーマットは次のとおりです。

ビット [31:13、11、9:4]

UNK/SBZP

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ExT, bit [12] External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

FS, bits [10,3:0]

Fault status bits. For the valid encodings of these bits in an ARMv7-A implementation with a VMSA, see Table B3-11 on page B3-50.

All encodings not shown in the table are reserved.

For information about using the IFSR see *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48.

Accessing the IFSR

To access the IFSR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c5, `<CRm>` set to c0, and `<opc2>` set to 1. For example:

```
MRC p15,0,<Rt>,c5,c0,1    ; Read CP15 Instruction Fault Status Register
MCR p15,0,<Rt>,c5,c0,1    ; Write CP15 Instruction Fault Status Register
```

c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)

The Auxiliary Data Fault Status Register (ADFSR) and the Auxiliary Instruction Fault Status Register (AIFSR) enable the system to return additional IMPLEMENTATION DEFINED fault status information, see *Auxiliary Fault Status Registers* on page B3-53.

The ADFSR and AIFSR are:

- 32-bit read/write registers
- accessible only in privileged modes
- when the Security Extensions are implemented, Banked registers
- introduced in ARMv7.

The formats of the ADFSR and AIFSR are IMPLEMENTATION DEFINED.

Accessing the ADFSR and AIFSR

To access the ADFSR or AIFSR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c5, `<CRm>` set to c1, and `<opc2>` set to:

- 0 for the ADFSR
- 1 for the AIFSR.

For example:

```
MRC p15,0,<Rt>,c5,c1,0    ; Read  CP15 Auxiliary Data Fault Status Register
MCR p15,0,<Rt>,c5,c1,0    ; Write CP15 Auxiliary Data Fault Status Register
MRC p15,0,<Rt>,c5,c1,1    ; Read  CP15 Auxiliary Instruction Fault Status Register
MCR p15,0,<Rt>,c5,c1,1    ; Write CP15 Auxiliary Instruction Fault Status Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ExT、ビット[12] 外部アボートタイプ。このビットは、外部アボートの実装定義の分類を提供するために使用できます。

外部アボート以外のアボートでは、このビットは常に 0 を返します。

FS、ビット [10、3:0]

フォールトステータス ビット。VMSA を含む ARMv7-A 実装における、これらのビットの有効なエンコードについては、P. B3-50 の表 B3-11 を参照して下さい。

この表に示されていないエンコードはすべて予約されています。

IFSR の使用方法については、*VMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B3-48 を参照して下さい。

IFSR へのアクセス

IFSR にアクセスするには、`<opc1>` を 0、`<CRn>` を c5、`<CRm>` を c0、`<opc2>` を 1 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c5,c0,1    ; Read CP15 Instruction Fault Status Register
MCR p15,0,<Rt>,c5,c0,1    ; Write CP15 Instruction Fault Status Register
```

c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)

補助データフォールト ステータスレジスタ (ADFSR) と補助命令フォールト ステータスレジスタ (AIFSR) は、システムが追加の実装定義のフォールトステータス情報を返すために使用できます。*補助フォールトステータス レジスタ* P. B3-53 を参照して下さい。

ADFSR と AIFSR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、バンクレジスタです。
- ARMv7 で導入されました。

ADFSR と AIFSR のフォーマットは実装定義です。

ADFSR と AIFSR へのアクセス

ADFSR または AIFSR にアクセスするには、`<opc1>` を 0、`<CRn>` を c5、`<CRm>` を c1、`<opc2>` を次の値に設定して、CP15 レジスタを読み出し / 書き込みします。

- 0（ADFSR の場合）
- 1（AIFSR の場合）

次に例を示します。

```
MRC p15,0,<Rt>,c5,c1,0    ; Read  CP15 Auxiliary Data Fault Status Register
MCR p15,0,<Rt>,c5,c1,0    ; Write CP15 Auxiliary Data Fault Status Register
MRC p15,0,<Rt>,c5,c1,1    ; Read  CP15 Auxiliary Instruction Fault Status Register
MCR p15,0,<Rt>,c5,c1,1    ; Write CP15 Auxiliary Instruction Fault Status Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.29 CP15 c6, Fault Address registers

There are two Fault Address registers, in CP15 c6, as shown in Figure B3-15 on page B3-120s. The two Fault Address registers complement the Fault Status registers, and are shown in Table B3-31.

Table B3-31 Fault Address registers

Register name	Description
Data Fault Address Register (DFAR)	*c6, Data Fault Address Register (DFAR)*
Instruction Fault Address Register (IFAR)	*c6, Instruction Fault Address Register (IFAR)* on page B3-125

Note

Before ARMv7:

- The DFAR was called the Fault Address Register (FAR).
- The Watchpoint Fault Address Register (DBGWFAR) was implemented in CP15 c6, with `<opc2>` = 1. From ARMv7, the DBGWFAR is only implemented as a CP14 debug register, see *Watchpoint Fault Address Register (DBGWFAR)* on page C10-28.

Fault information is returned using the fault address registers and the fault status registers described in *CP15 c5, Fault status registers* on page B3-121. For details of how these registers are used, and when the value in the IFAR is valid, see *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48.

c6, Data Fault Address Register (DFAR)

The Data Fault Address Register, DFAR, holds the MVA of the faulting address that caused a synchronous Data Abort exception.

The DFAR is:

- a 32-bit read/write register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Banked register.

The format of the DFAR is:

31 … 0
MVA of faulting address of synchronous Data Abort exception

For information about using the DFAR, and when the value in the DFAR is valid, see *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48.

A debugger can write to the DFAR to restore its value.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.12.29 CP15 c6、フォールトアドレス レジスタ

P. B3-120 の図 B3-15 に示すように、CP15 c6 には 2 つのフォールトアドレス レジスタがあります。これらの 2 つのフォールトアドレス レジスタは、表 B3-31 に示すように、フォールトステータス レジスタを補完します。

表 B3-31 フォールトアドレス レジスタ

レジスタ名	説明
データフォールト アドレスレジスタ (DFAR)	*c6、データフォールト アドレスレジスタ (DFAR)*
命令フォールト アドレスレジスタ (IFAR)	*c6、命令フォールト アドレスレジスタ (IFAR)* P. B3-125

注

ARMv7 以前については、次の注意事項があります。

- DFAR はフォールトアドレス レジスタ (FAR) と呼ばれていました。
- ウォッチポイントフォールト アドレスレジスタ (DBGWFAR) は、`<opc2>` = 1 の CP15 c6 に実装されていました。ARMv7 およびそれ以降では、DBGWFAR は CP14 デバッグレジスタとしてのみ実装されます（*ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10-28 を参照）。

フォールト情報は、フォールトアドレス レジスタと、*CP15 c5、フォールトステータス レジスタ* P. B3-121 で説明されているフォールトステータス レジスタを使用して返されます。これらのレジスタの使用方法の詳細と、IFAR の値が有効になるタイミングについては、*VMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B3-48 を参照して下さい。

c6、データフォールト アドレスレジスタ (DFAR)

データフォールト アドレスレジスタ (DFAR) には、同期データアボート例外を引き起こしたフォールトアドレスの MVA が保持されます。

DFAR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、バンクレジスタです。

DFAR のフォーマットは次のとおりです。

31 … 0
同期データアボート例外のフォールトアドレスの MVA

DFAR の使用方法と、DFAR の値が有効になるタイミングについては、*VMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B3-48 を参照して下さい。

デバッガは、DFAR に書き込みを行い、その値を復元することができます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the DFAR

To access the DFAR you read or write the CP15 registers with <opc1> set to 0, <CRn> set to c6, <CRm> set to c0, and <opc2> set to 0. For example:

```
MRC p15,0,<Rt>,c6,c0,0    ; Read CP15 Data Fault Address Register
MCR p15,0,<Rt>,c6,c0,0    ; Write CP15 Data Fault Address Register
```

c6, Instruction Fault Address Register (IFAR)

The Instruction Fault Address Register, IFAR, holds the MVA of the faulting access that caused a synchronous Prefetch Abort exception.

The IFAR is:

- a 32-bit read/write register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Banked register.

The format of the IFAR is:

31 … 0
MVA of faulting address of synchronous Prefetch Abort exception

For information about using the IFAR see *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48.

A debugger can write to the IFAR to restore its value.

Accessing the IFAR

To access the IFAR you read or write the CP15 registers with <opc1> set to 0, <CRn> set to c6, <CRm> set to c0, and <opc2> set to 2. For example:

```
MRC p15,0,<Rt>,c6,c0,2    ; Read CP15 Instruction Fault Address Register
MCR p15,0,<Rt>,c6,c0,2    ; Write CP15 Instruction Fault Address Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DFAR へのアクセス

DFAR にアクセスするには、<opc1> を 0、<CRn> を c6、<CRm> を c0、<opc2> を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c6,c0,0    ; Read CP15 Data Fault Address Register
MCR p15,0,<Rt>,c6,c0,0    ; Write CP15 Data Fault Address Register
```

c6、命令フォールト アドレスレジスタ (IFAR)

命令フォールト アドレスレジスタ (IFAR) には、同期プリフェッチアボート例外を引き起こしたフォールトアクセスの MVA が保持されます。

IFAR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、バンクレジスタです。

IFAR のフォーマットは次のとおりです。

31 … 0
同期プリフェッチアボート例外のフォールトアドレスの MVA

IFAR の使用方法については、*VMSA 実装のフォールト ステータス レジスタとフォールト アドレス レジスタ* P. B3- 48 を参照して下さい。

デバッガは、IFAR に書き込みを行い、その値を復元することができます。

IFAR へのアクセス

IFAR にアクセスするには、<opc1> を 0、<CRn> を c6、<CRm> を c0、<opc2> を 2 に設定して、CP15 レジスタを読み書きします。次に例を示します。

```
MRC p15,0,<Rt>,c6,c0,2    ; Read CP15 Instruction Fault Address Register
MCR p15,0,<Rt>,c6,c0,2    ; Write CP15 Instruction Fault Address Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.30 CP15 c7, Cache maintenance and other functions

The CP15 c7 registers are used for cache maintenance operations. They also provide barrier operations, and VA to PA address translation functions. Figure B3-16 shows the CP15 c7 registers.

Figure B3-16 CP15 c7 registers in a VMSA implementation

CP15 c7 register encodings not shown in Figure B3-16 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B3-69.

The CP15 c7 registers are described in:

- *CP15 c7, Cache and branch predictor maintenance functions*
- *CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130
- *CP15 c7, Miscellaneous functions* on page B3-136.

B3.12.31 CP15 c7, Cache and branch predictor maintenance functions

CP15 c7 provides a number of functions. This section describes only the CP15 c7 cache and branch predictor maintenance operations. Branch predictor operations are included in this section because they operate in a similar way to the cache maintenance operations.

Note

ARMv7 introduces significant changes in the CP c7 operations. Most of these changes are because, from ARMv7, the architecture covers multiple levels of cache. This section only describes the ARMv7 requirements for these operations. For details of these operations in previous versions of the architecture see:

- *c7, Cache operations* on page AppxG-38 for ARMv6
- *c7, Cache operations* on page AppxH-49 for ARMv4 and ARMv5.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.30 CP15 c7、キャッシュ保守機能とその他の機能

CP15 c7 レジスタは、キャッシュ保守操作に使用されます。これらのレジスタでは、バリア操作と VA から PA へのアドレス変換機能も提供されます。CP15 c7 レジスタを、図 B3-16 に示します。

図 B3-16 VMSA 実装の CP15 c7 レジスタ

図 B3-16 に示されていない CP15 c7 レジスタのエンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B3-69 を参照して下さい。

CP15 c7 レジスタについては、次のセクションで説明します。

- *CP15 c7、キャッシュおよび分岐予測器保守機能*
- *CP15 c7、仮想アドレスから物理アドレスへの変換操作* P. B3-130
- *CP15 c7、その他の機能* P. B3-136

B3.12.31 CP15 c7、キャッシュおよび分岐予測器保守機能

CP15 c7 では複数の機能が提供されます。このセクションでは、CP15 c7 のキャッシュおよび分岐予測器の保守操作についてのみ説明します。分岐予測器操作については、キャッシュ保守操作と似ているため、このセクションで説明します。

注

ARMv7 では、CP15 c7 操作が大幅に変更されました。これらの変更のほとんどは、ARMv7 からアーキテクチャで複数のキャッシュレベルがカバーされたことによります。このセクションでは、これらの操作に関する ARMv7 の要件についてのみ説明します。以前のバージョンの ARM アーキテクチャにおけるこれらの操作の詳細については、次のセクションを参照して下さい。

- *c7、キャッシュ操作* P. 付録 G-38（ARMv6 の場合）
- *c7、キャッシュ操作* P. 付録 H-49 （ARMv4 と ARMv5 の場合）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure B3-17 shows the CP15 c7 cache and branch predictor maintenance operations.

Figure B3-17 CP15 c7 Cache and branch predictor maintenance operations

The CP15 c7 cache and branch predictor maintenance operations are all write-only operations that can be executed only in privileged modes. They are listed in Table B3-32.

For more information about the terms used in this section see *Terms used in describing cache operations* on page B2-10. The Multiprocessing Extensions changes the set of caches affected by these operations, *Multiprocessor effects on cache maintenance operations* on page B2-23.

In Table B3-32, the *Rt data* column specifies what data is required in the register Rt specified by the `MCR` instruction used to perform the operation. For more information about the possible data formats, see *Data formats for the cache and branch predictor operations* on page B3-128.

Table B3-32 CP15 c7 cache and branch predictor maintenance operations

CRm	opc2	Mnemonic	Function[a]	Rt data
c1	0	ICIALLUIS[b]	Invalidate all instruction caches Inner Shareable to PoU. Also flushes branch target cache.[c]	Ignored
c1	6	BPIALLIS[b]	Invalidate entire branch predictor array Inner Shareable.	Ignored
c5	0	ICIALLU	Invalidate all instruction caches to PoU. Also flushes branch target cache.[c]	Ignored
c5	1	ICIMVAU	Invalidate instruction cache line by MVA to PoU.[c]	MVA
c5	6	BPIALL	Invalidate entire branch predictor array.	Ignored

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP15 c7 のキャッシュおよび分岐予測器の保守操作を、図 B3-17 に示します。

図 B3-17 CP15 c7 のキャッシュおよび分岐予測器の保守操作

CP15 c7 のキャッシュおよび分岐予測器の保守操作はすべて、特権モードでのみ実行可能な書き込み専用操作です。これらの操作の一覧を、表 B3-32 に示します。

このセクションで使用される用語の詳細については、*キャッシュ操作の説明で使用される用語* P. B2-10 を参照して下さい。マルチプロセッシング拡張機能が実装されている場合は、これらの操作の影響を受けるキャッシュのセットが異なります。*キャッシュの保守操作に対するマルチプロセッサの影響* P. B2-23 を参照して下さい。

表 B3-32 で、*Rt データ列*には、操作を実行するために使用される `MCR` 命令で指定されるレジスタ Rt に、どのデータが必要かが示されています。使用可能なデータフォーマットの詳細については、*キャッシュおよび分岐予測器の操作のデータフォーマット* P. B3-128 を参照して下さい。

表 B3-32 CP15 c7 のキャッシュおよび分岐予測器の保守操作

CRm	opc2	ニーモニック	機能[a]	Rt データ
c1	0	ICIALLUIS[b]	PoU まで、内部共有可能命令キャッシュ全体を無効化します。分岐先キャッシュのフラッシュも行います。[c]	無視
c1	6	BPIALLIS[b]	内部共有可能分岐予測器アレイ全体を無効化します。	無視
c5	0	ICIALLU	PoU まで、すべての命令キャッシュを無効化します。分岐先キャッシュのフラッシュも行います。[c]	無視
c5	1	ICIMVAU	PoU まで、MVA によって命令キャッシュラインを無効化します。[c]	MVA
c5	6	BPIALL	分岐予測器アレイ全体を無効化します。	無視

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-32 CP15 c7 cache and branch predictor maintenance operations (continued)

CRm	opc2	Mnemonic	Function[a]	Rt data
c5	7	BPIMVA	Invalidate MVA from branch predictor array.	MVA
c6	1	DCIMVAC	Invalidate data or unified cache line by MVA to PoC.	MVA
c6	2	DCISW	Invalidate data or unified cache line by set/way.	Set/way
c10	1	DCCMVAC	Clean data or unified cache line by MVA to PoC.	MVA
c10	2	DCCSW	Clean data or unified cache line by set/way.	Set/way
c11	1	DCCMVAU	Clean data or unified cache line by MVA to PoU.	MVA
c14	1	DCCIMVAC	Clean and Invalidate data or unified cache line by MVA to PoC.	MVA
c14	2	DCCISW	Clean and Invalidate data or unified cache line by set/way.	Set/way

a. *Modified Virtual Address* (MVA), *point of coherency* (PoC) and *point of unification* (PoU) are described in *Terms used in describing cache operations* on page B2-10.
b. Part of the Multiprocessing Extensions, See *Multiprocessor effects on cache maintenance operations* on page B2-23.
c. Only applies to separate instruction caches, does not apply to unified caches.

Data formats for the cache and branch predictor operations

Table B3-32 on page B3-127 shows three possibilities for the data in the register Rt specified by the `MCR` instruction. These are described in the following subsections:

- *Ignored*
- *MVA*
- *Set/way* on page B3-129.

Ignored

The value in the register specified by the `MCR` instruction is ignored. You do not have to write a value to the register before issuing the `MCR` instruction.

MVA

For more information about the possible meaning when the table shows that an MVA is required see *Terms used in describing cache operations* on page B2-10. When the data is stated to be an MVA, it does not have to be cache line aligned.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 B3-32 CP15 c7 のキャッシュおよび分岐予測器の保守操作（続き）

CRm	opc2	ニーモニック	機能[a]	Rt データ
c5	7	BPIMVA	分岐予測器アレイから MVA を無効化します。	MVA
c6	1	DCIMVAC	PoC まで、MVA によってデータまたは統一キャッシュラインを無効化します。	MVA
c6	2	DCISW	セット / ウェイによって、データまたは統一キャッシュラインを無効化します。	セット / ウェイ
c10	1	DCCMVAC	PoC まで、MVA によってデータまたは統一キャッシュラインをクリーニングします。	MVA
c10	2	DCCSW	セット / ウェイによって、データまたは統一キャッシュラインをクリーニングします。	セット / ウェイ
c11	1	DCCMVAU	PoU まで、MVA によってデータまたは統一キャッシュラインをクリーニングします。	MVA
c14	1	DCCIMVAC	PoC まで、MVA によってデータまたは統一キャッシュラインをクリーニングおよび無効化します。	MVA
c14	2	DCCISW	セット / ウェイによって、データまたは統一キャッシュラインをクリーニングおよび無効化します。	セット / ウェイ

a. *修飾仮想アドレス*(MVA)、*コヒーレンシ時点*(PoC)、および*統一時点*(PoU) については、*キャッシュ操作の説明で使用される用語* P. B2-10 で説明されています。
b. マルチプロセッシング拡張機能の一部です。*キャッシュの保守操作に対するマルチプロセッサの影響* P. B2-23 を参照して下さい。
c. 独立した命令キャッシュにのみ適用されます。統一キャッシュには適用されません。

キャッシュおよび分岐予測器の操作のデータフォーマット

`MCR` 命令によって指定されるレジスタ Rt に存在する可能性のある、3 種類のデータを P. B3-127 の表 B3-32 に示します。これらのデータについては、次のサブセクションで説明します。

- *無視*
- *MVA*
- *セット / ウェイ* P. B3-129

無視

`MCR` 命令で指定されたレジスタの値は無視されます。`MCR` 命令の発行前にレジスタに値を書き込む必要はありません。

MVA

表で MVA が必要であると示されている場合、その詳しい意味については、*キャッシュ操作の説明で使用される用語* P. B2-10 を参照して下さい。データが MVA であると示されている場合、その MVA はキャッシュラインにアラインしている必要はありません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Set/way

For an operation by set/way, the data identifies the cache line that the operation is to be applied to by specifying:

- the cache set the line belongs to
- the way number of the line in the set
- the cache level.

The format of the register data for a set/way operation is:

31 – 32-A	31-A – B	B-1 – L	L-1 – 4	3 – 1	0
Way	SBZ	Set	SBZ	Level	0

Where:

A = Log_2(ASSOCIATIVITY)

B = (L + S)

L = Log_2(LINELEN)

S = Log_2(NSETS)

ASSOCIATIVITY, LINELEN (Line Length) and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on.

The values of A and S are rounded up to the next integer.

Level ((Cache level to operate on) -1)

For example, this field is 0 for operations on L 1 cache, or 1 for operations on L 2 cache.

Set The number of the set to operate on.

Way The number of the way to operate on.

Note

- If L = 4 then there is no SBZ field between the set and level fields in the register.
- If A = 0 there is no way field in the register, and register bits [31:B] are SBZ.
- If the level, set or way field in the register is larger than the size implemented in the cache then the effect of the operation is UNPREDICTABLE.

Accessing the CP15 c7 cache and branch predictor maintenance operations

To perform one of the cache maintenance operations you write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c7, and `<CRm>` and `<opc2>` set to the values shown in Table B3-32 on page B3-127.

That is:

```
MCR p15,0,<Rt>,c7,<CRm>,<opc2>
```

For example:

```
MCR p15,0,<Rt>,c7,c5,0    ; Invalidate all instruction caches to point of unification
MCR p15,0,<Rt>,c7,c10,2   ; Clean data or unified cache line by set/way
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

セット／ウェイ

セット / ウェイによる操作の場合は、データで次の要素を指定することによって、操作が適用されるキャッシュラインが識別されます。

- ラインが属しているキャッシュセット
- セット内のラインのウェイ番号
- キャッシュレベル

セット / ウェイ操作のレジスタデータのフォーマットは次のとおりです。

31 – 32-A	31-A – B	B-1 – L	L-1 – 4	3 – 1	0
ウェイ	SBZ	セット	SBZ	レベル	0

各項目の説明については次を参照して下さい。

A = Log_2(ASSOCIATIVITY)

B = (L + S)

L = Log_2(LINELEN)

S = Log_2(NSETS)

ASSOCIATIVITY（アソシエティビティ）、LINELEN（ライン長）、NSETS（セット数）は、他の場所で使用されているのと同じ意味で、操作対象のキャッシュレベルに関する値です。

A と S の値は、それよりも大きい次の整数に丸められます。

レベル ((操作対象のキャッシュレベル) − 1)

例えば、このフィールドは、L1 キャッシュの操作では 0、L2 キャッシュの操作では 1 です。

セット 操作対象のセット番号

ウェイ 操作対象のウェイ番号

注

- L = 4 の場合は、レジスタのセットフィールドとレベルフィールドとの間に SBZ フィールドが存在しません。
- A = 0 の場合は、レジスタにウェイフィールドが存在せず、レジスタのビット [31:B] は SBZ です。
- レジスタのレベル、セット、またはウェイフィールドが、キャッシュに実装されているサイズより大きい場合、操作の影響は予測不能です。

CP15 c7 のキャッシュおよび分岐予測器の保守操作へのアクセス

キャッシュ保守操作のいずれかを実行するには、`<opc1>` を 0、`<CRn>` を c7、`<CRm>` と `<opc2>` を P. B3- 127 の表 B3- 32 に示す値に設定して、CP15 レジスタに書き込みます。

次の形式になります。

```
MCR p15,0,<Rt>,c7,<CRm>,<opc2>
```

次に例を示します。

```
MCR p15,0,<Rt>,c7,c5,0    ; Invalidate all instruction caches to point of unification
MCR p15,0,<Rt>,c7,c10,2   ; Clean data or unified cache line by set/way
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.32 CP15 c7, Virtual Address to Physical Address translation operations

CP15 c7 provides a number of functions, summarized in Figure B3-10 on page B3-65. This section describes only the CP15 c7 operations that provide *Virtual Address* (VA) to *Physical Address* (PA) translation on implementations that include the VMSA, and the register that returns the result of the operation.

Figure B3-18 shows all of the CP15 c7 VA to PA translation operations. It does not show the other CP15 c7 operations.

Note

As explained in this section, the CP15 c7 encodings for VA to PA translation with `<opc2>` == {4-7} are available only when the Security Extensions are implemented. These encodings are reserved and UNPREDICTABLE when the Security Extensions are not implemented.

Shown with Security Extensions implemented. When they are not implemented:
- the concepts of *Current security state* and *Other security state* are not defined
- encodings marked † are reserved and UNPREDICTABLE.

Figure B3-18 CP15 c7 VA to PA translation operations

This set of registers comprises:

- A single Physical Address Register, PAR, that returns the result of the VA to PA translation. For more information about this register see *c7, Physical Address Register (PAR) and VA to PA translations* on page B3-133.
- A set of VA to PA translation operations. These are:
  - — 32-bit write-only operations
  - — accessible only in privileged modes.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.32 CP15 c7、仮想アドレスから物理アドレスへの変換操作

CP15 c7 では、P. B3-65 の図 B3-10 に示す複数の機能が提供されます。このセクションでは、VMSA を含む実装で、*仮想アドレス*(VA) から *物理アドレス*(PA) への変換を提供する CP15 c7 操作と、操作の結果を返すレジスタについてのみ説明します。

CP15 c7 の VA から PA への変換操作のすべてを、図 B3-18 に示します。他の CP15 c7 操作は、この表には示されていません。

注

このセクションで説明するように、`<opc2>` = {4-7} を使用した VA から PA への変換に関する CP15 c7 エンコードは、セキュリティ拡張機能が実装されている場合にのみ使用できます。これらのエンコードは予約されており、セキュリティ拡張機能が実装されていない場合は予測不能です。

セキュリティ拡張機能が実装されている場合を示す。実装されていない場合は、次のようになる。
- 現在のセキュリティ状態とその他のセキュリティ状態の概念が定義されない。
- † マークされたエンコードが予約され、予測不能になる。

図 B3-18 CP15 c7 の VA から PA への変換操作

このレジスタのセットは、次のもので構成されます。

- VA から PA への変換の結果を返す単一の物理アドレスレジスタ (PAR)。このレジスタの詳細については、*c7、物理アドレスレジスタ (PAR) と VA から PA への変換* P. B3-133 を参照して下さい。
- VA から PA への変換操作のセット。これらには、次の特徴があります。
  - – 32 ビットの書き込み専用操作
  - – 特権モードでのみアクセス可能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When the Security Extensions are not implemented, there are four VA to PA translation operations, listed in Table B3-33.

Table B3-33 VA to PA translation when Security Extensions are not implemented

CRm	opc2	Mnemonic	Register or operation
c4	0	-	PAR, Physical Address Register
c8	0	V2PCWPR	Privileged read VA to PA translation
c8	1	V2PCWPW	Privileged write VA to PA translation
c8	2	V2PCWUR	User read VA to PA translation
c8	3	V2PCWUW	User write VA to PA translation

When the Security Extensions are implemented, there are eight VA to PA translation operations. Four of these are common to the Secure and Non-secure security states, and four are only available in the Secure state. Table B3-34 lists these operations, and shows the security states in which each is available.

Table B3-34 VA to PA translation when Security Extensions are implemented

			Register or operation:		
CRm	opc2	Mnemonic	Common	Non-secure state	Secure state
c4	0	-	-	PAR	PAR
c8	0	V2PCWPR	Current security state privileged read[a]	-	-
c8	1	V2PCWPW	Current security state privileged write[a]	-	-
c8	2	V2PCWUR	Current security state User read[a]	-	-
c8	3	V2PCWUW	Current security state User write[a]	-	-
c8	4	V2POWPR	-	-	Other security state privileged read[a]

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

セキュリティ拡張機能が実装されていない場合は、表 B3-33 に示す 4 つの VA から PA への変換操作があります。

表 B3-33 セキュリティ拡張機能が実装されていない場合の VA から PA への変換

CRm	opc2	ニーモニック	レジスタまたは操作
c4	0	–	PAR、物理アドレスレジスタ
c8	0	V2PCWPR	特権読み出し VA から PA への変換
c8	1	V2PCWPW	特権書き込み VA から PA への変換
c8	2	V2PCWUR	ユーザ読み出し VA から PA への変換
c8	3	V2PCWUW	ユーザ書き込み VA から PA への変換

セキュリティ拡張機能が実装されている場合は、8 つの VA から PA への変換操作があります。このうちの 4 つはセキュアと非セキュアのセキュリティ状態で共通であり、残りの 4 つはセキュア状態でのみ使用できます。これらの操作と、それぞれが使用可能なセキュリティ状態を、表 B3-34 に示します。

表 B3-34 セキュリティ拡張機能が実装されている場合の VA から PA への変換

			レジスタまたは操作		
CRm	opc2	ニーモニック	共通	非セキュア状態	セキュア状態
c4	0	–	–	PAR	PAR
c8	0	V2PCWPR	現在のセキュリティ状態の特権読み出し[a]	–	–
c8	1	V2PCWPW	現在のセキュリティ状態の特権書き込み[a]	–	–
c8	2	V2PCWUR	現在のセキュリティ状態のユーザ読み出し[a]	–	–
c8	3	V2PCWUW	現在のセキュリティ状態のユーザ書き込み[a]	–	–
c8	4	V2POWPR	–	–	他のセキュリティ状態の特権読み出し[a]

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-34 VA to PA translation when Security Extensions are implemented (continued)

Register or operation:					
CRm	**opc2**	**Mnemonic**	**Common**	**Non-secure state**	**Secure state**
c8	5	V2POWPW	-	-	Other security state privileged write[a]
c8	6	V2POWUR	-	-	Other security state User read[a]
c8	7	V2POWUW	-	-	Other security state User write[a]

a. VA to PA Translation operations.

Writing a VA to a VA to PA translation operation encoding translates the VA to the corresponding PA. The PA value is returned in the PAR. These operations are accessible only in privileged modes. The available VA to PA translations depend on:

- whether the Security Extensions are implemented
- if the Security Extensions are implemented, whether the processor is in the Secure or Non-secure state.

In more detail:

Security Extensions not implemented

Four VA to PA translation operations are available, as shown in Table B3-33 on page B3-131. These operations provide VA to PA translation for privileged read or write, and for User read or write.

Security Extensions implemented, processor in Non-secure state

Only the four current security state VA to PA translation operations are available, as shown in Table B3-33 on page B3-131. These operations provide VA to PA translation for privileged read or write, and for User read or write, in the Non-secure security state.

It is not possible to perform VA to PA translations for the Secure security state. Attempting to access an Other security state VA to PA translation operation encoding generates an Undefined Instruction exception.

Security Extensions implemented, processor in Secure security state

Eight VA to PA Translation operations are available, as shown in Table B3-34 on page B3-131:

- The four current security state VA to PA translation operations provide address translation for privileged read or write, and for User read or write, in the Secure security state.
- The four other security state VA to PA translation operations provide address translation for privileged read or write, and for User read or write, in the Non-secure security state.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 B3-34 セキュリティ拡張機能が実装されている場合の VA から PA への変換（続き）

レジスタまたは操作					
CRm	**opc2**	**ニーモニック**	**共通**	**非セキュア状態**	**セキュア状態**
c8	5	V2POWPW	–	–	他のセキュリティ状態の特権書き込み[a]
c8	6	V2POWUR	–	–	他のセキュリティ状態のユーザ読み出し[a]
c8	7	V2POWUW	–	–	他のセキュリティ状態のユーザ書き込み[a]

a. VA から PA への変換操作

VA から PA への変換操作エンコードに VA を書き込むと、VA が対応する PA に変換されます。PA の値は PAR に返されます。これらの操作には、特権モードでのみアクセスできます。使用可能な VA から PA への変換は、次の要素によって異なります。

- セキュリティ拡張機能が実装されているかどうか。
- セキュリティ拡張機能が実装されている場合、プロセッサがセキュア状態か非セキュア状態か。

詳細は次のとおりです。

セキュリティ拡張機能が実装されていない場合

P. B3-131 の表 B3-33 に示す 4 つの VA から PA への変換操作を使用できます。これらの操作によって、特権読み出しまたは書き込み用と、ユーザ読み出しまたは書き込み用の VA から PA への変換が提供されます。

セキュリティ拡張機能が実装されており、プロセッサが非セキュア状態の場合

P. B3-131 の表 B3-33 に示されている、4 つの、現在のセキュリティ状態の VA から PA への変換操作のみを使用できます。これらの操作によって、非セキュアセキュリティ状態での特権読み出しまたは書き込み用と、ユーザ読み出しまたは書き込み用の VA から PA への変換が提供されます。

セキュアセキュリティ状態での VA から PA への変換は実行できません。他のセキュリティ状態の VA から PA への変換操作エンコードにアクセスを試みると、未定義命令例外が発生します。

セキュリティ拡張機能が実装されており、プロセッサがセキュアセキュリティ状態の場合

P. B3-131 の表 B3-34 に示すように、8 つの VA から PA への変換操作を使用できます。

- 4 つの、現在のセキュリティ状態の VA から PA への変換操作では、セキュアセキュリティ状態における特権読み出しまたは書き込み用、とユーザ読み出しまたは書き込み用のアドレス変換が提供されます。
- 4 つの、他のセキュリティ状態の VA から PA への変換操作では、非セキュアセキュリティ状態における特権読み出しまたは書き込み用と、ユーザ読み出しまたは書き込み用のアドレス変換が提供されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

In all cases:

- If the FCSE is implemented the VA required is the VA before any modification by the FCSE, not the MVA.
- For information about translations when the MMU is disabled see *VA to PA translation when the MMU is disabled* on page B3-136.

c7, Physical Address Register (PAR) and VA to PA translations

The *Physical Address Register*, PAR, of the current security state receives the PA during any VA to PA translation.

The PAR is:

- a 32-bit read/write register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Banked register.

Write access to the register means its contents can be context switched.

The PAR format depends on the value of bit [0]. Bit [0] indicates whether the address translation operation completed successfully.

If the translation completed successfully, the format of the PAR is:

PA, bits [31:12]

Physical Address. The physical address corresponding to the supplied virtual address. Address bits [31:12] are returned.

Bit [11] Reserved. UNK/SBZP.

Bits [10:1] Return information from the translation table entry used for the translation:

NOS, bit [10]

Not Outer Shareable attribute. Indicates whether the physical memory is Outer Shareable:

0 Memory is Outer Shareable

1 Memory is not Outer Shareable.

On an implementation that do not support Outer Shareable, this bit is UNK/SBZP.

NS, bit [9] Non-secure. The NS bit from the translation table entry.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

すべての場合について、次の規則が適用されます。

- FCSE が実装されている場合に必要な VA は、MVA ではなく FCSE による変更前の VA です。
- MMU が禁止されている場合の変換の詳細については、*MMU が非稼動の場合の VA から PA への変換* P. B3- 136 を参照して下さい。

c7、物理アドレスレジスタ (PAR) と VA から PA への変換

現在のセキュリティ状態の*物理アドレスレジスタ*(PAR) は、VA から PA への変換時に PA を受け取ります。

PAR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、バンクレジスタです。

このレジスタへの書き込みアクセスは、その内容がコンテキストスイッチ可能であることを意味します。

PAR のフォーマットはビット [0] の値によって異なります。ビット [0] は、アドレス変換操作が正常に完了したかどうかを示します。

変換が正常に完了した場合の PAR のフォーマットは次のとおりです。

PA、ビット [31:12]

物理アドレス。提供された仮想アドレスに対応する物理アドレス。アドレスのビット [31:12] が返されます。

ビット [11] 予約。UNK/SBZP

ビット [10:1] 変換に使用された変換テーブルエントリからの情報を返します。

NOS、ビット [10]

非外部共有可能属性。物理メモリが外部共有可能かどうかを示します。

0 メモリが外部共有可能です。

1 メモリが外部共有不可です。

外部共有可能がサポートされていない実装では、このビットは UNK/SBZP です。

NS、ビット [9]

非セキュア。変換テーブルエントリからの NS ビット。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Bit [8] IMPLEMENTATION DEFINED.

SH, bit [7]

Shareable attribute. Indicates whether the physical memory is Shareable:

0 Memory is Non-shareable

1 Memory is Shareable.

Inner[2:0], bits [6:4]

Inner memory attributes from the translation table entry. Permitted values are:

0b111 Write-Back, no Write-Allocate

0b110 Write-Through

0b101 Write-Back, Write-Allocate

0b011 Device

0b001 Strongly-ordered

0b000 Non-cacheable.

Other encodings for Inner[2:0] are reserved.

Outer[1:0], bits [3:2]

Outer memory attributes from the translation table. Possible values are:

0b11 Write-Back, no Write-Allocate.

0b10 Write-Through, no Write-Allocate.

0b01 Write-Back, Write-Allocate.

0b00 Non-cacheable.

SS, bit [1] SuperSection. Used to indicate if the result is a Supersection:

0 Page is not a Supersection, that is, PAR[31:12] contains PA[31:12], regardless of the page size.

1 Page is part of a Supersection

- PAR[31:24] contains PA[31:24]
- PAR[23:16] contains PA[39:32]
- PAR[15:12] contains 0b0000.

If an implementation supports less than 40 bits of physical address, the bits in the PAR field that correspond to physical address bits that are not implemented are UNKNOWN.

Note

PA[23:12] is the same as VA[23:12] for Supersections

F, bit [0] F bit is 0 if the conversion completed successfully.

In the Inner[2:0] and Outer[1:0] fields, an implementation that does not support all of the attributes can report the memory type behavior that the cache does support, rather than the value held in the translation table entry.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ビット [8] 実装定義

SH、ビット [7]

共有可能属性。物理メモリが共有可能かどうかを示します。

0 メモリが共有不可です。

1 メモリが共有可能です。

Inner[2:0]、ビット [6:4]

変換テーブルエントリからの内部メモリ属性。許される値は次のとおりです。

0b111 ライトバック、書き込み割り当てなし

0b110 ライトスルー

0b101 ライトバック、書き込み割り当て

0b011 デバイス

0b001 ストロングリオーダ

0b000 キャッシュ不可

Inner[2:0] の他のエンコードは予約されています。

Outer[1:0]、ビット [3:2]

変換テーブルエントリからの外部メモリ属性。許される値は次のとおりです。

0b11 ライトバック、書き込み割り当てなし

0b10 ライトスルー、書き込み割り当てなし

0b01 ライトバック、書き込み割り当て

0b00 キャッシュ不可

SS、ビット [1]

スーパーセクション。結果がスーパーセクションであるかどうかを示すために使用されます。

0 ページはスーパーセクションではありません。つまり、ページサイズにかかわらず、PAR[31:12] に PA[31:12] が含まれています。

1 ページはスーパーセクションの一部です。

- PAR[31:24] に PA[31:24] が含まれています。
- PAR[23:16] に PA[39:32] が含まれています。
- PAR[15:12] に 0b0000 が含まれています。

実装でサポートしている物理アドレスが 40 ビット未満の場合、実装されていない物理アドレスに対応する PAR フィールドのビットは未知です。

注

PA[23:12] はスーパーセクションの VA[23:12] と同じです。

F、ビット [0] F ビットは、変換が正常に完了した場合は 0 です。

すべての属性をサポートしていない実装では、Inner[2:0] フィールドと Outer[1:0] フィールドを使用して、変換テーブルエントリに保持された値ではなく、キャッシュでサポートされているメモリタイプの動作を報告することができます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If the translation fails without generating an abort, the format of the PAR is:

31 … 7	6 … 1	0
UNK/SBZP	FS	F

Bits [31:7] UNK/SBZP.

FS, bits [6:1] Fault status bits. Bits [12,10,3:0] from the Data Fault Status Register, indicate the source of the abort. For more information, see *c5, Data Fault Status Register (DFSR)* on page B3-121.

F, bit [0] F bit is 1 if the conversion aborted.

The VA to PA translation only generates an abort if the translation fails because an external abort occurred on a translation table walk request. In this case:

- If the external abort is synchronous, the DFSR and DFAR of the security state in which the abort is handled are updated. The DFSR indicates the appropriate external abort on Translation fault, and the DFAR indicates the MVA that caused the translation. PAR is UNKNOWN.
- If the external abort is asynchronous, the DFSR of the security state in which the abort is handled is updated when the abort is taken. The DFSR indicates the asynchronous external abort. The DFAR is not updated. PAR is UNKNOWN.

For all other cases where the VA to PA translation fails:

- No abort is generated, and the DFSR and DFAR are unchanged
- the PAR [6:1] field is updated with an FSR encoding that indicates the fault
- the PAR bit [0] is set to 1.

Implementations that do not support all attributes can report the behavior for those memory types that the cache does support.

Accessing the PAR and the VA to PA translation operations

To access one of the VA to PA translation operations you write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c7, `<CRm>` set to c8, and `<opc2>` set to the value shown in Table B3-33 on page B3-131 or Table B3-34 on page B3-131.

With register `Rt` containing the original VA this gives:

```
MCR p15,0,<Rt>,c7,c8,<opc2>
```

To read the PAR you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c7, `<CRm>` set to c4, and `<opc2>` set to 0. To return the translated PA in register Rt this gives:

```
MRC p15,0,<Rt>,c7,c4,0
```

The PAR is a read/write register, and you can write to the CP15 registers with the same settings to write to the register. There is no translation operation that requires writing to this register, but the write operation might be required to restore the value of the PAR after a context switch.

An example of a VA to PA translation when the Security Extensions are not implemented is:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アボートが発生せずに変換が失敗した場合、PAR のフォーマットは次のとおりです。

31 … 7	6 … 1	0
UNK/SBZP	FS	F

ビット [31:7] UNK/SBZP

FS、ビット[6:1] フォールトステータス ビット。データフォールト ステータスレジスタからのビット [12、10、3:0] で、アボートのソースを示します。詳細については、*c5、データフォールト ステータスレジスタ (DFSR)* P. B3-121 を参照して下さい。

F、ビット[0] F ビットは、変換がアボートされた場合は 1 です。

変換テーブルウォーク要求で外部アボートが発生したため、VA から PA への変換が失敗した場合にのみ、アボートが発生します。この場合、次の動作が行われます。

- 外部アボートが同期の場合は、アボートが処理されるセキュリティ状態の DFSR と DFAR が更新されます。DFSR は変換フォールトでの該当する外部アボートを示し、DFAR はその変換を引き起こした MVA を示します。PAR は未知です。
- 外部アボートが非同期の場合は、アボートが処理されるセキュリティ状態の DFSR が、アボートの取得時に更新されます。DFSR では、非同期外部アボートが示されます。DFAR は更新されません。PAR は未知です。

VA から PA への変換が失敗する他のすべての場合は、次の処理が行われます。

- アボートは生成されず、DFSR と DFAR は変更されません。
- PAR[6:1] フィールドは、フォールトを示す FSR エンコードで更新されます。
- PAR ビット [0] は 1 にセットされます。

すべての属性をサポートしない実装では、キャッシュでサポートされているメモリタイプの動作を報告することができます。

PAR 操作と VA から PA への変換操作へのアクセス

VA から PA への変換操作のいずれかにアクセスするには、`<opc1>` を 0、`<CRn>` を c7、`<CRm>` を c8、`<opc2>` を P. B3-131 の表 B3-33 または P. B3-131 の表 B3-34 に示す値に設定して、CP15 レジスタに書き込みます。

レジスタ `Rt` が元の VA を含む場合、次のようになります。

```
MCR p15,0,<Rt>,c7,c8,<opc2>
```

PAR を読み出すには、`<opc1>` を 0、`<CRn>` を c7、`<CRm>` を c4、`<opc2>` を 0 に設定して、CP15 レジスタを読み出します。変換された PA をレジスタ Rt に返すには、次のようになります。

```
MRC p15,0,<Rt>,c7,c4,0
```

PAR は読み出し / 書き込みレジスタで、このレジスタに書き込むための設定は、CP15 レジスタに書き込む場合と同じです。このレジスタへの書き込みが必要な変換操作はありませんが、コンテキストスイッチ後に PAR の値を復元するため、書き込み操作が必要になる可能性があります。

セキュリティ拡張機能が実装されていない場合の、VA から PA への変換例を次に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
MCR p15,0,<Rt>,c7,c8,2    ; Write CP15 VA to User Read VA to PA Translation Register
MRC p15,0,<Rt>,c7,c4,0    ; Read  CP15 PA from Physical Address Register
```

An example of a VA to PA translation when the Security Extensions are implemented and the processor is in the Secure state is:

```
MCR p15,0,<Rt>,c7,c8,5    ; Write VA to Other State Privileged Write VA to PA Translation Register
                          ; Performs VA to PA translation for Non-secure security state
MRC p15,0,<Rt>,c7,c4,0    ; Read PA from Physical Address Register
```

VA to PA translation when the MMU is disabled

The VA to PA translation operations occur even when the MMU is disabled. The operations report the flat address mapping and the MMU-disabled value of the attributes and permissions for the data side accesses. These include any MMU-disabled re-mapping specified by the TEX-remap facilities. The SuperSection bit is 0 when the MMU is disabled. For more information about the address and attributes returned when the MMU is disabled see *Enabling and disabling the MMU* on page B3-5.

When the Security Extensions are implemented, this information applies when the MMU is disabled in the security state for which the VA to PA translation is performed.

B3.12.33 CP15 c7, Miscellaneous functions

CP15 c7 provides a number of functions, summarized in Figure B3-10 on page B3-65. This section describes only the CP15 c7 miscellaneous operations.

Figure B3-19 shows the CP15 c7 miscellaneous operations. It does not show the other CP15 c7 operations.

Figure B3-19 CP15 c7 Miscellaneous operations

The CP15 c7 miscellaneous operations are described in:

- *CP15 c7, Data and Instruction Barrier operations* on page B3-137
- *CP15 c7, No Operation (NOP)* on page B3-138.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
MCR p15,0,<Rt>,c7,c8,2    ; Write CP15 VA to User Read VA to PA Translation Register
MRC p15,0,<Rt>,c7,c4,0    ; Read  CP15 PA from Physical Address Register
```

セキュリティ拡張機能が実装されており、プロセッサがセキュア状態である場合の、VA から PA への変換例を次に示します。

```
MCR p15,0,<Rt>,c7,c8,5    ; Write VA to Other State Privileged Write VA to PA Translation Register
                          ; Performs VA to PA translation for Non-secure security state
MRC p15,0,<Rt>,c7,c4,0    ; Read PA from Physical Address Register
```

MMU が非稼動の場合の VA から PA への変換

MMU が非稼動の場合にも、VA から PA への変換操作は実行されます。この操作では、フラット アドレスマッピングと、データ側アクセス用の属性とアクセス許可の、MMU が非稼動のときの値が報告されます。これらには、TEX 再マップ機能で指定された、MMU が非稼動のときの再マッピングが含まれます。MMU が非稼動のときは、スーパーセクションビットは 0 です。MMU が非稼動のときに返されるアドレスと属性の詳細については、*MMU の許可と禁止* P. B3-5 を参照して下さい。

セキュリティ拡張機能が実装されている場合は、この情報は、VA から PA への変換が実行されるセキュリティ状態で MMU が非稼動の場合に適用されます。

B3.12.33 CP15 c7、その他の機能

CP15 c7 では、P. B3-65 の図 B3-10 に示す複数の機能が提供されます。このセクションでは、CP15 c7 のその他の操作についてのみ説明します。

CP15 c7 のその他の操作を、図 B3-19 に示します。他の CP15 c7 操作は、この表には示されていません。

図 B3-19 CP15 c7、その他の操作

CP15 c7 のその他の操作については、次のサブセクションで説明します。

- *CP15 c7、データおよび命令バリア操作* P. B3-137
- *CP15 c7、無操作 (NOP)* P. B3-138

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP15 c7, Data and Instruction Barrier operations

ARMv6 includes two CP15 c7 operations to perform Data Barrier operations, and another operation to perform an Instruction Barrier operation. In ARMv7:

- The ARM and Thumb instruction sets include instructions to perform the barrier operations, that can be executed in all modes, see *Memory barriers* on page A3-47.
- The CP15 c7 operations are defined as write-only operations, that can be executed in all modes. The three operations are described in:
  - — *Instruction Synchronization Barrier operation*
  - — *Data Synchronization Barrier operation*
  - — *Data Memory Barrier operation*.

  The value in the register Rt specified by the `MCR` instruction used to perform one of these operations is ignored. You do not have to write a value to the register before issuing the `MCR` instruction.

  In ARMv7 using these CP15 c7 operations is deprecated. Use the `ISB`, `DSB`, and `DMB` instructions instead.

Note

- In ARMv6 and earlier documentation, the Instruction Synchronization Barrier operation is referred to as a Prefetch Flush (PFF).
- In versions of the ARM architecture before ARMv6 the Data Synchronization Barrier operation is described as a *Data Write Barrier* (DWB).

Instruction Synchronization Barrier operation

In ARMv7, the `ISB` instruction is used to perform an Instruction Synchronization Barrier, see *ISB* on page A8-102.

The deprecated CP15 c7 encoding for an Instruction Synchronization Barrier is `<opc1>` set to 0, `<CRn>` set to c7, `<CRm>` set to c5, and `<opc2>` set to 4.

Data Synchronization Barrier operation

In ARMv7, the `DSB` instruction is used to perform a Data Synchronization Barrier, see *DSB* on page A8-92.

The deprecated CP15 c7 encoding for a Data Synchronization Barrier is `<opc1>` set to 0, `<CRn>` set to c7, `<CRm>` set to c10, and `<opc2>` set to 4. This operation performs the full system barrier performed by the `DSB` instruction.

Data Memory Barrier operation

In ARMv7, the `DMB` instruction is used to perform a Data Memory Barrier, see *DMB* on page A8-90.

The deprecated CP15 c7 encoding for a Data Memory Barrier is `<opc1>` set to 0, `<CRn>` set to c7, `<CRm>` set to c10, and `<opc2>` set to 5. This operation performs the full system barrier performed by the `DMB` instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP15 c7、データおよび命令バリア操作

ARMv6 には、データバリア操作を実行するための 2 つの CP15 c7 操作と、命令バリア操作を実行するための別の操作が含まれています。ARMv7 では、次の規則が適用されます。

- ARM 命令セットと Thumb 命令セットには、すべてのモードで実行可能なバリア操作を実行するための命令が含まれています（*メモリバリア* P. A3-47 を参照）。
- CP15 c7 操作は、すべてのモードで実行可能な書き込み専用操作として定義されます。これらの 3 つの操作については、次のサブセクションで説明します。
  - – *命令同期バリア操作*
  - – *データ同期バリア操作*
  - – *データ メモリバリア操作*

  これらのいずれかの操作を実行するための `MCR` 命令で指定されたレジスタ Rt の値は無視されます。`MCR` 命令の発行前にレジスタに値を書き込む必要はありません。

  ARMv7 では、これらの CP15 c7 操作の使用は非推奨です。代わりに、`ISB`、`DSB`、`DMB` 命令を使用して下さい。

注

- ARMv6 およびそれ以前の説明書では、命令同期バリア操作がプリフェッチフラッシュ (PFF) と呼ばれています。
- ARMv6 以前のバージョンの ARM アーキテクチャでは、データ同期バリア操作がデータ*書き込みバリア*(DWB) と呼ばれています。

命令同期バリア操作

ARMv7 では、命令同期バリアを実行するために `ISB` 命令が使用されます。*ISB* P. A8-102 を参照して下さい。

命令同期バリア用の CP15 c7 エンコードは、`<opc1>` を 0、`<CRn>` を c7、`<CRm>` を c5、`<opc2>` を 4 に設定しますが、このエンコードは非推奨です。

データ同期バリア操作

ARMv7 では、データ同期バリアを実行するために `DSB` 命令が使用されます。*DSB* P. A8-92 を参照して下さい。

データ同期バリア用の CP15 c7 エンコードは、`<opc1>` を 0、`<CRn>` を c7、`<CRm>` を c10、`<opc2>` を 4 に設定しますが、このエンコードは非推奨です。この操作では、`DSB` 命令で実行される完全なシステムバリアが実行されます。

データ メモリバリア操作

ARMv7 では、データ メモリバリアを実行するために `DMB` 命令が使用されます。*DMB* P. A8-90 を参照して下さい。

データ メモリバリア用の CP15 c7 エンコードは、`<opc1>` を 0、`<CRn>` を c7、`<CRm>` を c10、`<opc2>` を 5 に設定しますが、このエンコードは非推奨です。この操作では、`DMB` 命令で実行される完全なシステムバリアが実行されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP15 c7, No Operation (NOP)

ARMv6 includes two CP15 c7 operations that are not supported in ARMv7, with encodings that become *No Operation* (NOP) in ARMv7. These are:

- The *Wait For Interrupt* (CP15WFI) operation. In ARMv7 this operation is performed by the `WFI` instruction, that is available in the ARM and Thumb instruction sets. For more information, see *WFI* on page A8-810.
- The prefetch instruction by MVA operation. In ARMv7 this operation is replaced by the `PLI` instruction, that is available in the ARM and Thumb instruction sets. For more information, see *PLI (immediate, literal)* on page A8-242 and *PLI (register)* on page A8-244.

In ARMv7, the CP15 c7 encodings that were used for these operations must be valid write-only operations that perform a NOP. These encodings are:

- for the ARMv6 CP15WFI operation:
  - `<opc1>` set to 0, `<CRn>` set to c7, `<CRm>` set to c0, and `<opc2>` set to 4
- for the ARMv6 prefetch instruction by MVA operation:
  - `<opc1>` set to 0, `<CRn>` set to c7, `<CRm>` set to c13, and `<opc2>` set to 1.

B3.12.34 CP15 c8, TLB maintenance operations

On ARMv7-A implementations, CP15 c8 operations are used for TLB maintenance functions. Figure B3-20 shows the CP15 c8 encodings.

Figure B3-20 CP15 c8 operations

CP15 c8 encodings not shown in Figure B3-20 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B3-69.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP15 c7、無操作 (NOP)

ARMv6 には、ARMv7 では*無操作*(NOP) になるエンコードを使用した、ARMv7 ではサポートされていない 2 つの CP15 c7 操作が含まれています。具体的には、次のものがあります。

- *割り込み待ち*(CP15WFI) 操作。ARMv7 では、この操作が、ARM および Thumb 命令セットで使用可能な `WFI` 命令によって実行されます。詳細については、*WFI* P. A8-810 を参照して下さい。
- MVA による命令のプリフェッチ操作。ARMv7 では、この操作が、ARM および Thumb 命令セットで使用可能な `PLI` 命令に置き換えられます。詳細については、*PLI (イミディエート、リテラル)* P. A8-242 と *PLI (レジスタ)* P. A8-244 を参照して下さい。

ARMv7 では、これらの操作用に使用されていた CP15 c7 エンコードを、NOP を実行する有効な書き込み専用操作にする必要があります。これらのエンコードは次のとおりです。

- ARMv6 CP15WFI 操作の場合
  - `<opc1>` を 0、`<CRn>` を c7、`<CRm>` を c0、`<opc2>` を 4 に設定
- ARMv6 MVA による命令のプリフェッチ操作の場合
  - `<opc1>` を 0、`<CRn>` を c7、`<CRm>` を c13、`<opc2>` を 1 に設定

B3.12.34 CP15 c8、TLB 保守操作

ARMv7-A 実装では、CP15 c8 操作は TLB 保守機能に使用されます。CP15 c8 エンコードを、図 B3-20 に示します。

図 B3-20 CP15 c8 操作

図 B3-20 に示されていない CP15 c8 エンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B3-69 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The CP15 c8 TLB maintenance functions:

- are write-only operations
- can be executed only in privileged modes.

Table B3-35 summarizes the TLB maintenance operations.

Table B3-35 CP15 c8 TLB maintenance operations

CRm	opc2	Mnemonic	Function	Rt data
c3	0	TLBIALLIS	Invalidate entire unified TLB[d] Inner Shareable[a]	Ignored
	1	TLBIMVAIS	Invalidate unified TLB[d] entry by MVA Inner Shareable[a]	MVA
	2	TLBIASIDIS	Invalidate unified TLB[d] by ASID match Inner Shareable[a]	ASID
	3	TLBIMVAAIS	Invalidate unified TLB[d] entry by MVA all ASID Inner Shareable[a]	MVA
c5	0	ITLBIALL	Invalidate entire instruction TLB[b]	Ignored
	1	ITLBIMVA	Invalidate instruction TLB[b] entry by MVA	MVA
	2	ITLBIASID	Invalidate instruction TLB[b] by ASID match	ASID
c6	0	DTLBIALL	Invalidate entire data TLB[b]	Ignored
	1	DTLBIMVA	Invalidate data TLB[b] entry by MVA	MVA
	2	DTLBIASID	Invalidate data TLB[b] by ASID match	ASID
c7	0	TLBIALL[c]	Invalidate entire unified TLB[d]	Ignore
	1	TLBIMVA[c]	Invalidate unified TLB[d] entry by MVA	MVA
	2	TLBIASID[c]	Invalidate unified TLB[d] by ASID match	ASID
	3	TLBIMVAA	Invalidate unified TLB[d] entries by MVA all ASID[a]	MVA

a. Implemented only as part of the Multiprocessing Extensions.
b. If these operations are performed on an implementation that has a unified TLB they operate on the unified TLB.
c. The mnemonics for the operations with CRm==c7, opc2=={0,1,2} were previously UTLBIALL, UTLBIMVA and UTLBIMASID.
d. When separate instruction and data TLBs are implemented, these operations are performed on both TLBs.

For more information about the Inner Shareable operations see *Multiprocessor effects on TLB maintenance operations* on page B3-62.

For information about the effect of these operations on locked TLB entries see *The interaction of TLB maintenance operations with TLB lockdown* on page B3-57.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP15 c8 TLB 保守機能には、次の特徴があります。

- 書き込み専用操作
- 特権モードでのみ実行できます。

TLB 保守操作を、表 B3-35 に示します。

表 B3-35 CP15 c8 TLB 保守操作

CRm	opc2	ニーモニック	機能	Rt データ
c3	0	TLBIALLIS	内部共有可能[a] 統一 TLB[d] 全体の無効化	無視
	1	TLBIMVAIS	内部共有可能[a]MVA による統一 TLB[d] エントリの無効化	MVA
	2	TLBIASIDIS	内部共有可能[a]ASID 一致による統一 TLB[d] の無効化	ASID
	3	TLBIMVAAIS	内部共有可能[a]MVA とすべての ASID による統一 TLB[d] エントリの無効化	MVA
c5	0	ITLBIALL	命令 TLB[b] 全体の無効化	無視
	1	ITLBIMVA	MVA による命令 TLB[b] エントリの無効化	MVA
	2	ITLBIASID	ASID 一致による命令 TLB[b] の無効化	ASID
c6	0	DTLBIALL	データ TLB[b] 全体の無効化	無視
	1	DTLBIMVA	MVA によるデータ TLB[b] エントリの無効化	MVA
	2	DTLBIASID	ASID 一致によるデータ TLB[b] の無効化	ASID
c7	0	TLBIALL[c]	統一 TLB[d] 全体の無効化	無視
	1	TLBIMVA[c]	MVA による統一 TLB[d] エントリの無効化	MVA
	2	TLBIASID[c]	ASID 一致による統一 TLB[d] の無効化	ASID
	3	TLBIMVAA	MVA とすべての ASID[a] による統一 TLB[d] エントリの無効化	MVA

a. マルチプロセッシング拡張機能の一部としてのみ実装されます。
b. これらの操作が統一 TLB を含む実装で実行された場合は、その統一 TLB が対象になります。
c. CRm==c7 と opc2=={0,1,2} を使用した操作のニーモニックは、以前は UTLBIALL、UTLBIMVA、および UTLBIMASID でした。
d. 独立した命令 TLB とデータ TLB が実装されている場合は、これらの操作が両方の TLB に対して実行されます。

内部共有可能操作の詳細については、*TLB 保守操作に対するマルチプロセッサの影響* P. B3-62 を参照して下さい。

ロックされた TLB エントリに対するこれらの操作の影響については、*TLB 保守操作と TLB ロックダウンとの相互動作* P. B3-57 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

About the TLB maintenance operations

For more information about TLBs and their maintenance see *Translation Lookaside Buffers (TLBs)* on page B3-54, and in particular *TLB maintenance* on page B3-56. The following subsections give more information about the TLB maintenance operations:

- *Invalidate entire TLB*
- *Invalidate single TLB entry by MVA*
- *Invalidate TLB entries by ASID match*
- *Invalidate TLB entries by MVA all ASID* on page B3-141.

As stated in the footnotes to Table B3-35 on page B3-139:

- If an Instruction TLB or Data TLB operation is used on a system that implements a Unified TLB then the operation is performed on the Unified TLB
- If a Unified TLB operation is used on a system that implements separate Instruction and Data TLBs then the operation is performed on both the Instruction TLB and the Data TLB.
- The mnemonics for the operations to invalidate a unified TLB that are defined in the ARM v7 base architecture were previously UTLBIALL, UTLBIMV, and UTLBIASID. These remain synonyms for these operations, but ARM deprecates the use of the older names. These are the operations with CRm==c7, opc2=={0,1,2}.

For information about the synchronization of the TLB maintenance operations see *TLB maintenance operations and the memory order model* on page B3-59.

Invalidate entire TLB

The Invalidate entire TLB operations invalidate all unlocked entries in the TLB. The value in the register Rt specified by the `MCR` instruction used to perform the operation is ignored. You do not have to write a value to the register before issuing the `MCR` instruction.

Invalidate single TLB entry by MVA

The Invalidate Single Entry operations invalidate a TLB entry that matches the MVA and ASID values provided as an argument to the operation. The register format required is:

31 … 12	11 … 8	7 … 0
MVA	SBZ	ASID

With global entries in the TLB, the supplied ASID value is not checked.

Invalidate TLB entries by ASID match

The Invalidate on ASID Match operations invalidate all TLB entries for non-global pages that match the ASID value provided as an argument to the operation. The register format required is:

31 … 8	7 … 0
SBZ	ASID

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

TLB 保守操作について

TLB とその保守の詳細については、*変換ルックアサイドバッファ (T L B)* P. B3-54、特に *TLB 保守* P. B3-56 を参照して下さい。次のサブセクションでは、TLB 保守操作についてより詳しく説明します。

- *TLB 全体の無効化*
- *MVA による単一 TLB エントリの無効化*
- *ASID 一致による TLB エントリの無効化*
- *MVA とすべての ASID による TLB エントリの無効化* P. B3-141

P. B3-139 の表 B3-35 の脚注に記載されている内容について

- 命令 TLB またはデータ TLB の操作が、統一 TLB を実装しているシステムで使用された場合は、その操作が統一 TLB に対して実行されます。
- 統一 TLB 操作が、独立した命令 TLB とデータ TLB を実装しているシステムで使用された場合は、その操作が命令 TLB とデータ TLB の両方に対して実行されます。
- ARM v7 基本アーキテクチャで定義されている、統一 TLB を無効化するための操作のニーモニックは、以前は UTLBIALL、UTLBIMV、および UTLBIASID でした。これらのニーモニックは、この操作の同義語として残っていますが、古い名前の使用はお勧めできません。現在の操作では、CRm==c7、opc2=={0,1,2} が使用されます。

TLB 保守操作の同期については、*TLB 保守操作とメモリオーダ モデル* P. B3-59 を参照して下さい。

TLB 全体の無効化

TLB 全体の無効化操作では、TLB 内のロックされていないすべてのエントリが無効化されます。この操作を実行するための `MCR` 命令で指定されたレジスタ Rt の値は無視されます。`MCR` 命令の発行前にレジスタに値を書き込む必要はありません。

MVA による単一 TLB エントリの無効化

単一エントリの無効化操作では、引数として操作に渡された MVA および ASID の値と一致する、TLB エントリが無効化されます。必要なレジスタフォーマットは次のとおりです。

31 … 12	11 … 8	7 … 0
MVA	SBZ	ASID

TLB のグローバルエントリでは、渡された ASID 値はチェックされません。

ASID 一致による TLB エントリの無効化

ASID 一致での無効化操作では、引数として操作に渡された ASID 値と一致する、非グローバルページのすべての TLB エントリが無効化されます。必要なレジスタフォーマットは次のとおりです。

31 … 8	7 … 0
SBZ	ASID

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Invalidate TLB entries by MVA all ASID

The Invalidate TLB entries by MVA all ASID operations invalidate all TLB entries that matches the MVA provided as an argument to the operation regardless of the ASID. The register format required is:

31 … 12	11 … 0
MVA	SBZ

Accessing the CP15 c8 TLB maintenance operations

To perform one of the TLB maintenance operations you write the CP15 registers with `<opc1> == 0`, `<CRn>==c8`, and `<CRm>` and `<opc2>` set to the values shown in Table B3-35 on page B3-139. That is:

```
MCR p15,0,<Rt>,c8,<CRm>,<opc2>
```

For example:

```
MCR p15,0,<Rt>,c8,c5,0    ; Invalidate all unlocked entries in Instruction TLB
MCR p15,0,<Rt>,c8,c6,2    ; Invalidate Data TLB entries on ASID match
```

B3.12.35 CP15 c9, Cache and TCM lockdown registers and performance monitors

Some CP15 c9 encodings are reserved for IMPLEMENTATION DEFINED memory system functions, in particular:

- cache control, including lockdown
- TCM control, including lockdown
- branch predictor control.

Additional CP15 c9 encodings are reserved for performance monitors. These encodings fall into two groups:

- the optional performance monitors, described in Chapter C9 *Performance Monitors*
- additional IMPLEMENTATION DEFINED performance monitors.

The reserved encodings permit implementations that are compatible with previous versions of the ARM architecture, in particular with the ARMv6 requirements. Figure B3-21 shows the permitted CP15 c9 register encodings.

Figure B3-21 Permitted CP15 c9 encodings

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

MVA とすべての ASID による TLB エントリの無効化

MVA とすべての ASID による TLB エントリの無効化操作では、ASID にかかわらず、引数として操作に渡された MVA と一致するすべての TLB エントリが無効化されます。必要なレジスタフォーマットは次のとおりです。

31 … 12	11 … 0
MVA	SBZ

CP15 c8 TLB 保守操作へのアクセス

TLB 保守操作のいずれかを実行するには、`<opc1> == 0`、`<CRn>== c8`、`<CRm>` と `<opc2>` を P. B3- 139 の表 B3- 35 に示す値に設定して、CP15 レジスタに書き込みます。次の形式になります。

```
MCR p15,0,<Rt>,c8,<CRm>,<opc2>
```

次に例を示します。

```
MCR p15,0,<Rt>,c8,c5,0    ; Invalidate all unlocked entries in Instruction TLB
MCR p15,0,<Rt>,c8,c6,2    ; Invalidate Data TLB entries on ASID match
```

B3.12.35 CP15 c9、キャッシュおよび TCM ロックダウンレジスタとパフォーマンスモニタ

一部の CP15 c9 エンコードが、実装定義のメモリシステム機能、特に次の機能のため予約されています。

- ロックダウンを含むキャッシュ制御
- ロックダウンを含む TCM 制御
- 分岐予測器の制御

追加の CP15 c9 エンコードが、パフォーマンスモニタ用として予約されています。これらのエンコードは、次の 2 つのグループに分類されます。

- 第 C9 章「パフォーマンスモニタ」で説明されている、オプションのパフォーマンスモニタ
- 追加の実装定義のパフォーマンスモニタ

予約されているエンコードでは、特に ARMv6 要件を伴う、以前のバージョンの ARM アーキテクチャと互換性のある実装が許可されます。許可される CP15 c9 レジスタのエンコードを、図 B3- 21 に示します。

図 B3- 21 許可される CP15 c9 エンコード

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP15 c9 encodings not shown in Figure B3-21 on page B3-141 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B3-69.

In ARMv6, CP15 c9 provides cache lockdown functions. With the ARMv7 abstraction of the hierarchical memory model, for CP15 c9:

- All encodings with CRm = {c0-c2, c5-c8} are reserved for IMPLEMENTATION DEFINED cache, branch predictor and TCM operations.

  This reservation enables the implementation of a scheme that is backwards compatible with ARMv6. For details of the ARMv6 implementation see *c9, Cache lockdown support* on page AppxG-45.

  Note

  In an ARMv6 implementation that implements the Security Extensions, a Cache Behavior Override Register is required in CP15 c9, with CRm = 8, see *c9, Cache Behavior Override Register (CBOR)* on page AppxG-49. This register is not architecturally-defined in ARMv7, and therefore the CP15 c9 encoding with CRm = 8 is IMPLEMENTATION DEFINED. However, an ARMv7 implementation can include the CBOR, in which case ARM recommends that this encoding is used for it.

- All encodings with CRm = {c12-c14} are reserved for the optional performance monitors that are defined in Chapter C9 *Performance Monitors*.

- All encodings with CRm = c15 are reserved for IMPLEMENTATION DEFINED performance monitoring features.

B3.12.36 CP15 c10, Memory remapping and TLB control registers

On ARMv7-A implementations, CP15 c10 is used for memory remapping registers. In addition, some encodings are reserved for IMPLEMENTATION DEFINED TLB control functions, in particular TLB lockdown. The reserved encodings permit implementations that are compatible with previous versions of the ARM architecture, in particular with the ARMv6 requirements.

Figure B3-22 shows the CP15 c10 registers and reserved encodings.

Figure B3-22 CP15 c10 registers

CP15 c10 encodings not shown in Figure B3-22 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B3-69.

The CP15 c10 memory remap registers are described in *CP15 c10, Memory Remap Registers* on page B3-143.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

P. B3-141 の図 B3-21 に示されていない CP15 c9 エンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B3-69 を参照して下さい。

ARMv6 では、CP15 c9 によってキャッシュロックダウン機能が提供されます。階層型メモリモデルの ARMv7 抽象化を使用した CP15 c9 の場合、次の規則が適用されます。

- CRm == {c0-c2,c5-c8} を使用したすべてのエンコードは、実装定義のキャッシュ、分岐予測器、および TCM 操作用として予約されています。

  この予約によって、ARMv6 と下位互換性のある方式の実装が可能になります。ARMv6 実装の詳細については、*c9、キャッシュロックダウンのサポート* P. 付録 G-45 を参照して下さい。

  注

  セキュリティ拡張機能を含む ARMv6 実装では、CP15 c9 に、CRm = 8 のキャッシュ動作オーバライドレジスタが必要です。*c9、キャッシュ動作オーバライドレジスタ (CBOR)* P. 付録 G-49 を参照して下さい。このレジスタは ARMv7 ではアーキテクチャ定義でないため、CRm = 8 の CP15 c9 エンコードは実装定義です。ただし、ARMv7 実装に CBOR を含めることはできます。その場合は、このエンコードを CBOR に使用することをお勧めします。

- CRm = {c12-c14} であるすべてのエンコードは、第 C9 章 *「パフォーマンスモニタ」* で定義されているオプションのパフォーマンスモニタ用として予約されています。

- CRm = c15 であるすべてのエンコードは、実装定義のパフォーマンスモニタ機能用として予約されています。

B3.12.36 CP15 c10、メモリ再マッピングおよび TLB 制御レジスタ

ARMv7-A 実装では、CP15 c10 が、メモリ再マッピングレジスタに使用されます。さらに、一部のエンコードは実装定義の TLB 制御機能、特に TLB ロックダウン用として予約されています。予約されているエンコードでは、特に ARMv6 要件を伴う、以前のバージョンの ARM アーキテクチャと互換性のある実装が許可されます。

CP15 c10 レジスタと、予約されているエンコードを図 B3-22 に示します。

図 B3-22 CP15 c10 レジスタ

図 B3-22 に示されていない CP15 c10 エンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B3-69 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The IMPLEMENTATION DEFINED TLB control operations

In VMSAv6, CP15 c10 provides TLB lockdown functions. In VMSAv7, the TLB lockdown mechanism is IMPLEMENTATION DEFINED and some CP15 c10 encodings are reserved for IMPLEMENTATION DEFINED TLB control operations. These are the encodings with `<CRn>` == c10, `<opc1>` == 0, `<CRm>` == {c0, c1, c4, c8}, and `<opc2>` == {0-7}.

B3.12.37 CP15 c10, Memory Remap Registers

CP15 c10 includes two Memory Remap Registers, described in the subsections:

- *c10, Primary Region Remap Register (PRRR)*
- *c10, Normal Memory Remap Register (NMRR)* on page B3-146.

In addition:

- The significance and use of these registers is described in *Memory region attribute descriptions when TEX remap is enabled* on page B3-34.
- The function of these registers is architecturally defined only when the
  - — SCTLR.TRE bit is set to 1
  - — SCTLR.TRE bit is set to 0 and no IMPLEMENTATION DEFINED mechanism using MMU remap has been invoked.

  Otherwise their behavior is IMPLEMENTATION DEFINED, see *SCTLR.TRE, SCTLR.M, and the effect of the MMU remap registers* on page B3-38.

c10, Primary Region Remap Register (PRRR)

The Primary Region Remap Register, PRRR, can in some cases control the top level mapping of the TEX[0], C, and B memory region attributes.

The PRRR:

- is a 32-bit read/write register
- is accessible only in privileged modes
- when the Security Extensions are implemented:
  - — is a Banked register
  - — has write access to the Secure copy of the register disabled when the **CP15SDISABLE** signal is asserted HIGH.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP15 c10 メモリ再マップレジスタについては、*CP15 c10、メモリ再マップレジスタ* P. B3-143 で説明します。

実装定義の TLB 制御操作

VMSAv6 では、CP15 c10 によって TLB ロックダウン機能が提供されます。VMSAv7 では、TLB ロックダウン機構は実装定義で、一部の CP15 c10 エンコードは実装定義の TLB 制御操作用として予約されています。これらのエンコードは、`<CRn>` == c10、`<opc1>` == 0、`<CRm>` == {c0, c1, c4, c8}、`<opc2>` == {0-7} のものです。

B3.12.37 CP15 c10、メモリ再マップレジスタ

CP15 c10 には、次のサブセクションで説明されている 2 つのメモリ再マップレジスタが含まれています。

- *c10、1 次領域再マップレジスタ (PRRR)*
- *c10、ノーマルメモリ再マップレジスタ (NMRR)* P. B3-146

また、次の説明が適用されます。

- これらのレジスタの意味と用途については、*TEX 再マップが許可されている場合のメモリ領域の属性の説明* P. B3-34 で説明します。
- これらのレジスタの機能は、次の場合のみアーキテクチャ定義です。
  - – SCTLR.TRE ビットが 1 にセットされている。
  - – SCTLR.TRE ビットが 0 にセットされ、MMU 再マップを使用した実装定義の機構が呼び出されていない。

  それ以外の場合、動作は実装定義です。*SCTLR.TRE、SCTLR.M、および MMU 再マップレジスタの効果* P. B3-38 を参照して下さい。

c10、1 次領域再マップレジスタ (PRRR)

1 次領域再マップレジスタ (PRRR) を使用して、TEX[0]、C、B のメモリ領域属性の最上位マッピングを制御できる場合があります。

PRRR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、次の特徴があります。
  - – バンクレジスタ
  - – レジスタのセキュアコピーへの書き込みアクセスは、CP15SDISABLE 信号が HIGH にアサートされているときには不可能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the PRRR is:

The reset value of the PRRR is IMPLEMENTATION DEFINED.

NOS*n*, bit [24+*n*], for values of *n* from 0 to 7

Outer Shareable property mapping for memory attributes *n*, if the region is mapped as Normal Memory that is Shareable. *n* is the value of the TEX[0], C and B bits, see Table B3-36 on page B3-145. The possible values of each NOS*n* bit are:

0 Shareable Normal memory region is Outer Shareable

1 Shareable Normal Memory region is Inner Shareable.

The value of this bit is ignored if the region is not Shareable Normal memory.

The meaning of the field with $n = 6$ is IMPLEMENTATION DEFINED and might differ from the meaning given here. This is because the meaning of the attribute combination {TEX[0] = 1, C = 1, B = 0} is IMPLEMENTATION DEFINED.

If the implementation does not support the Outer Shareable memory attribute then these bits are reserved, RAZ/SBZP.

Bits [23:20] Reserved. UNK/SBZP.

NS1, bit [19] Mapping of S = 1 attribute for Normal memory. This bit gives the mapped Shareable attribute for a region of memory that:

- is mapped as Normal memory
- has the S bit set to 1.

The possible values of the bit are:

0 Region is not Shareable

1 Region is Shareable.

NS0, bit [18] Mapping of S = 0 attribute for Normal memory. This bit gives the mapped Shareable attribute for a region of memory that:

- is mapped as Normal memory
- has the S bit set to 0.

The possible values of the bit are the same as those given for the NS1 bit, bit [19].

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

PRRR のフォーマットは次のとおりです。

PRRR のリセット時の値は実装定義です。

NOS*n*、ビット [24+*n*]、*n* の値が 0 ～ 7 の場合

領域が共有可能なノーマルメモリとしてマッピングされている場合の、メモリ属性 *n* の外部共有可能プロパティマッピング。*n* は、TEX[0]、C、B ビットの値です。P. B3-145 の表 B3-36 を参照して下さい。各 NOS*n* ビットの可能な値は次のとおりです。

0 共有可能ノーマルメモリ領域が外部共有可能です。

1 共有可能ノーマルメモリ領域が内部共有可能です。

このビットの値は、領域が共有可能ノーマルメモリでない場合は無視されます。

$n = 6$ のフィールドの意味は実装定義で、ここに示されている意味とは異なる場合があります。これは、属性の組み合わせ {TEX[0] = 1, C = 1, B = 0} の意味が実装定義であるためです。

実装で外部共有可能メモリ属性がサポートされていない場合は、これらのビットは予約済みで、RAZ/SBZP です。

ビット [23:20] 予約。UNK/SBZP

NS1、ビット [19]

ノーマルメモリの S = 1 属性のマッピング。このビットによって、次のようなメモリ領域について、マッピングされた共有可能属性が提供されます。

- ノーマルメモリとしてマッピングされている。
- S ビットが 1 にセットされている。

このビットの可能な値は次のとおりです。

0 領域が共有不可です。

1 領域が共有可能です。

NS0、ビット [18]

ノーマルメモリの S = 0 属性のマッピング。このビットによって、次のようなメモリ領域について、マッピングされた共有可能属性が提供されます。

- ノーマルメモリとしてマッピングされている。
- S ビットが 0 にセットされている。

このビットの可能な値は、NS1 ビット（ビット [19]）の値と同じです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DS1, bit [17] Mapping of S = 1 attribute for Device memory. This bit gives the mapped Shareable attribute for a region of memory that:

- is mapped as Device memory
- has the S bit set to 1.

The possible values of the bit are the same as those given for the NS1 bit, bit [19].

DS0, bit [16] Mapping of S = 0 attribute for Device memory. This bit gives the mapped Shareable attribute for a region of memory that:

- is mapped as Device memory
- has the S bit set to 0.

The possible values of the bit are the same as those given for the NS1 bit, bit [19].

TR*n*, bits [2*n*+1:2*n*] for values of *n* from 0 to 7

Primary TEX mapping for memory attributes *n*. *n* is the value of the TEX[0], C and B bits, see Table B3-36. This field defines the mapped memory type for a region with attributes *n*. The possible values of the field are:

00 Strongly-ordered

01 Device

10 Normal Memory

11 Reserved, effect is UNPREDICTABLE.

The meaning of the field with *n* = 6 is IMPLEMENTATION DEFINED and might differ from the meaning given here. This is because the meaning of the attribute combination {TEX[0] = 1, C = 1, B = 0} is IMPLEMENTATION DEFINED.

Table B3-36 shows the mapping between the memory region attributes and the *n* value used in the PRRR.nOS*n* and PRRR.TR*n* field descriptions.

Table B3-36 Memory attributes and the *n* value for the PRRR field descriptions

Attributes			*n* value
TEX[0]	C	B	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DS1、ビット [17]

デバイスメモリの S = 1 属性のマッピング。このビットによって、次のようなメモリ領域について、マッピングされた共有可能属性が提供されます。

- デバイスメモリとしてマッピングされている。
- S ビットが 1 にセットされている。

このビットの可能な値は、NS1 ビット（ビット [19]）の値と同じです。

DS0、ビット [16]

デバイスメモリの S = 0 属性のマッピング。このビットによって、次のようなメモリ領域について、マッピングされた共有可能属性が提供されます。

- デバイスメモリとしてマッピングされている。
- S ビットが 0 にセットされている。

このビットの可能な値は、NS1 ビット（ビット [19]）の値と同じです。

TR*n*、ビット [2*n*+1:2*n*]、*n* の値が 0 ～ 7 の場合

メモリ属性 *n* の 1 次 TEX マッピング。*n* は、TEX[0]、C、B ビットの値です。表 B3-36 を参照して下さい。このフィールドでは、属性が *n* の領域について、マッピングされたメモリタイプが定義されます。このフィールドの可能な値は次のとおりです。

00 ストロングリオーダ

01 デバイス

10 ノーマルメモリ

11 予約済み、効果は予測不能

n = 6 のフィールドの意味は実装定義で、ここに示されている意味とは異なる場合があります。これは、属性の組み合わせ {TEX[0] = 1, C = 1, B = 0} の意味が実装定義であるためです。

メモリ領域属性と、PRRR.nOS*n* および PRRR.TR*n* のフィールド記述で使用される *n* の値とのマッピングを、表 B3-36 に示します。

表 B3-36 メモリ属性と PRRR フィールド記述の *n* の値

属性			*n* の値
TEX[0]	C	B	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B3-36 Memory attributes and the *n* value for the PRRR field descriptions (continued)

Attributes			*n* value
TEX[0]	C	B	
1	1	0	6
1	1	1	7

For more information about the PRRR see *Memory region attribute descriptions when TEX remap is enabled* on page B3-34.

Accessing the PRRR

To access the PRRR you read or write the CP15 registers with <opc1> set to 0, <CRn> set to c10, <CRm> set to c2, and <opc2> set to 0. For example:

```
MRC p15,0,<Rt>,c10,c2,0    ; Read  CP15 Primary Region Remap Register
MCR p15,0,<Rt>,c10,c2,0    ; Write CP15 Primary Region Remap Register
```

c10, Normal Memory Remap Register (NMRR)

The Normal Memory Remap Register, NMRR, can in some cases provide additional mapping controls for memory regions that are mapped as Normal memory by their entry in the PRRR.

The NMRR:

- is a 32-bit read/write register
- is accessible only in privileged modes
- when the Security Extensions are implemented:
  - — is a Banked register
  - — has write access to the Secure copy of the register disabled when the **CP15SDISABLE** signal is asserted HIGH.

The format of the NMRR is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OR7		OR6		OR5		OR4		OR3		OR2		OR1		OR0		IR7		IR6		IR5		IR4		IR3		IR2		IR1		IR0	

The reset value of the NMRR is IMPLEMENTATION DEFINED.

OR*n*, bits [2*n*+17:2*n*+16], for values of *n* from 0 to 7

Outer Cacheable property mapping for memory attributes *n*, if the region is mapped as Normal Memory by the TR*n* entry in the PRRR, see *c10, Primary Region Remap Register (PRRR)* on page B3-143. *n* is the value of the TEX[0], C and B bits, see Table B3-36 on page B3-145. The possible values of this field are:

00 Region is Non-cacheable

01 Region is Write-Back, WriteAllocate

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B3-36 メモリ属性と PRRR フィールド記述の *n* の値（続き）

属性			*n* の値
TEX[0]	C	B	
1	1	0	6
1	1	1	7

PRRR の詳細については、*TEX 再マップが許可されている場合のメモリ領域の属性の説明* P. B3-34 を参照して下さい。

PRRR へのアクセス

PRRR にアクセスするには、<opc1> を 0、<CRn> を c10、<CRm> を c2、<opc2> を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c10,c2,0    ; Read  CP15 Primary Region Remap Register
MCR p15,0,<Rt>,c10,c2,0    ; Write CP15 Primary Region Remap Register
```

c10、ノーマルメモリ再マップレジスタ (NMRR)

ノーマルメモリ再マップレジスタ (NMRR) を使用して、PRRR のエントリによってノーマルメモリとしてマッピングされたメモリ領域の、追加のマッピング制御を提供できる場合があります。

NMRR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、次の特徴があります。
  - – バンクレジスタ
  - – レジスタのセキュアコピーへの書き込みアクセスは、CP15SDISABLE 信号が HIGH にアサートされているときには不可能です。

NMRR のフォーマットは次のとおりです

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OR7		OR6		OR5		OR4		OR3		OR2		OR1		OR0		IR7		IR6		IR5		IR4		IR3		IR2		IR1		IR0	

NMRR のリセット時の値は実装定義です。

OR*n*、ビット [2*n* + 17:2*n* + 16]、*n* の値が 0 ～ 7 の場合

領域が PRRR の TR*n* エントリによってノーマルメモリとしてマッピングされている場合の、メモリ属性 *n* の外部キャッシュ可能プロパティマッピング。*c10、1 次領域再マップレジスタ (PRRR)* P. B3-143 を参照して下さい。*n* は、TEX[0]、C、B ビットの値です。P. B3-145 の表 B3-36 を参照して下さい。このフィールドの可能な値は次のとおりです。

00 領域がキャッシュ不可です。

01 領域がライトバック、書き込み割り当てです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

10 Region is WriteThrough, Non-WriteAllocate

11 Region is Write-Back, Non-WriteAllocate.

The meaning of the field with n = 6 is IMPLEMENTATION DEFINED and might differ from the meaning given here. This is because the meaning of the attribute combination {TEX[0] = 1, C = 1, B = 0} is IMPLEMENTATION DEFINED.

IR*n*, bits [2*n*+1:2*n*], for values of *n* from 0 to 7

Inner Cacheable property mapping for memory attributes *n*, if the region is mapped as Normal Memory by the TR*n* entry in the PRRR, see *c10, Primary Region Remap Register (PRRR)* on page B3-143. *n* is the value of the TEX[0], C and B bits, see Table B3-36 on page B3-145. The possible values of this field are the same as those given for the OR*n* field.

The meaning of the field with n = 6 is IMPLEMENTATION DEFINED and might differ from the meaning given here. This is because the meaning of the attribute combination {TEX[0] = 1, C = 1, B = 0} is IMPLEMENTATION DEFINED.

For more information about the NMRR see *Memory region attribute descriptions when TEX remap is enabled* on page B3-34.

Accessing the NMRR

To access the NMRR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c10, `<CRm>` set to c2, and `<opc2>` set to 1. For example:

```
MRC p15,0,<Rt>,c10,c2,1    ; Read  CP15 Normal Memory Remap Register
MCR p15,0,<Rt>,c10,c2,1    ; Write CP15 Normal Memory Remap Register
```

B3.12.38 CP15 c11, Reserved for TCM DMA registers

Some CP15 c11 register encodings are reserved for IMPLEMENTATION DEFINED DMA operations to and from TCM, see Figure B3-23:

Figure B3-23 Permitted CP15 c11 encodings

CP15 c11 encodings not shown in Figure B3-23 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B3-69.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

10 領域がライトスルー、書き込み割り当てなしです。

11 領域がライトバック、書き込み割り当てなしです。

n = 6 のフィールドの意味は実装定義で、ここで示されている意味とは異なる場合があります。これは、属性の組み合わせ {TEX[0] = 1, C = 1, B = 0} の意味が実装定義のためです。

IR*n*、ビット [2*n* + 1:2*n*]、*n* の値が 0 ～ 7 の場合

領域が PRRR 内の TR*n* エントリによってノーマルメモリとしてマッピングされている場合の、メモリ属性 *n* の内部キャッシュ可能プロパティマッピング。*c10、1 次領域再マップレジスタ (PRRR)* P. B3-143 を参照して下さい。*n* は、TEX[0]、C、B ビットの値です。P. B3-145 の表 B3-36 を参照して下さい。このフィールドの可能な値は、OR*n* フィールドのものと同じです。

n = 6 のフィールドの意味は実装定義で、ここで示されている意味とは異なる場合があります。これは、属性の組み合わせ {TEX[0] = 1, C = 1, B = 0} の意味が実装定義のためです。

NMRR の詳細については、*TEX 再マップが許可されている場合のメモリ領域の属性の説明* P. B3-34 を参照して下さい。

NMRR へのアクセス

NMRR にアクセスするには、`<opc1>` を 0、`<CRn>` を c10、`<CRm>` を c2、`<opc2>` を 1 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c10,c2,1    ; Read  CP15 Normal Memory Remap Register
MCR p15,0,<Rt>,c10,c2,1    ; Write CP15 Normal Memory Remap Register
```

B3.12.38 CP15 c11、TCM DMA 用に予約されているレジスタ

一部の CP15 c11 レジスタのエンコードは、TCM に対する実装定義の DMA 操作用として予約されています。図 B3-23 を参照して下さい。

図 B3-23 許可される CP15 c11 エンコード

図 B3-23 に示されていない CP15 c11 エンコードは予測不能です。*未割り当ての CP15 エンコード* P. B3-69 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.39 CP15 c12, Security Extensions registers

When the Security Extensions are implemented, CP15 c12 is used for the Vector base address registers and an Interrupt status register. Figure B3-24 shows the CP15 c12 Security Extensions registers:

Figure B3-24 Security Extensions CP15 c12 registers

When the Security Extensions are implemented, CP15 c12 encodings not shown in Figure B3-24 are UNPREDICTABLE. On an implementation that does not include the Security Extensions all CP15 c12 encodings are UNDEFINED. For more information, see *Unallocated CP15 encodings* on page B3-69.

The CP15 c12 registers are described in the subsections:

- *c12, Vector Base Address Register (VBAR)*
- *c12, Monitor Vector Base Address Register (MVBAR)* on page B3-149
- *c12, Interrupt Status Register (ISR)* on page B3-150.

B3.12.40 c12, Vector Base Address Register (VBAR)

When the Security Extensions are implemented and high exception vectors are not selected, the Vector Base Address Register, VBAR, provides the exception base address for exceptions that are not handled in Monitor mode, see *Exception vectors and the exception base address* on page B1-30. The high exception vectors always have the base address `0xFFFF0000` and are not affected by the value of VBAR.

The VBAR:

- Is present only when the Security Extensions are implemented.
- Is a 32-bit read/write register.
- Is accessible only in privileged modes.
- Has a defined reset value, for the Secure copy of the register, of 0. This reset value does not apply to the Non-secure copy of the register, and software must program the Non-secure copy of the register with the required value, as part of the processor boot sequence.
- Is a Banked register.
- Has write access to the Secure copy of the register disabled when the **CP15SDISABLE** signal is asserted HIGH.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.12.39 CP15 c12、セキュリティ拡張機能レジスタ

セキュリティ拡張機能が実装されている場合は、CP15 c12 がベクタ ベースアドレスレジスタと割り込みステータスレジスタに使用されます。CP15 c12 セキュリティ拡張機能レジスタを、図 B3-24 に示します。

図 B3-24 セキュリティ拡張機能 CP15 c12 レジスタ

セキュリティ拡張機能が実装されている場合、図 B3-24 に示されていない CP15 c12 エンコードは予測不能です。セキュリティ拡張機能を含まない実装では、すべての CP15 c12 エンコードは未定義です。詳細については、*未割り当ての CP15 エンコード* P. B3-69 を参照して下さい。

CP15 c12 レジスタについては、次のサブセクションで説明します。

- *c12、ベクタ ベースアドレスレジスタ (VBAR)*
- *c12、モニタベクタ ベースアドレス レジスタ (MVBAR)* P. B3-149
- *c12、割り込みステータスレジスタ (ISR)* P. B3-150

B3.12.40 c12、ベクタ ベースアドレスレジスタ (VBAR)

セキュリティ拡張機能が実装され、上位例外ベクタが選択されていない場合は、ベクタ ベースアドレスレジスタ (VBAR) によって、モニタモードで処理されない例外の例外ベースアドレスが提供されます。*例外ベクタと例外ベースアドレス* P. B1-30 を参照して下さい。上位例外ベクタは常にベースアドレスが `0xFFFF0000` で、VBAR の値の影響を受けません。

VBAR の特徴は次のとおりです。

- セキュリティ拡張機能が実装されている場合にのみ存在します。
- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- レジスタのセキュアコピーは、リセット時の値が 0 として定義されています。このリセット時の値は、レジスタの非セキュアコピーには適用されないため、ソフトウェアでプロセッサのブートシーケンスの一部として、レジスタの非セキュアコピーを必要な値にプログラムする必要があります。
- バンクレジスタ
- レジスタのセキュアコピーへの書き込みアクセスは、CP15SDISABLE 信号が HIGH にアサートされたときに禁止されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the VBAR is:

31 … 5	4 … 0
Vector_Base_Address	(0) (0) (0) (0) (0)

The Secure copy of the VBAR holds the vector base address for the Secure state, described as the Secure exception base address

The Non-secure copy of the VBAR holds the vector base address for the Non-secure state, described as the Non-secure exception base address.

Vector_Base_Address, bits [31:5]

Bits [31:5] of the base address of the normal exception vectors. Bits [4:0] of an exception vector is the exception offset, see Table B1-3 on page B1-31.

Bits [4:0] Reserved, UNK/SBZP.

For details of how the VBAR registers are used to determine the exception addresses see *Exception vectors and the exception base address* on page B1-30.

Accessing the VBAR

To access the VBAR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c12, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c12,c0,0    ; Read  CP15 Vector Base Address Register
MCR p15,0,<Rt>,c12,c0,0    ; Write CP15 Vector Base Address Register
```

B3.12.41 c12, Monitor Vector Base Address Register (MVBAR)

The Monitor Vector Base Address Register, MVBAR, provides the exception base address for all exceptions that are handled in Monitor mode, see *Exception vectors and the exception base address* on page B1-30.

The MVBAR is:

- present only when the Security Extensions are implemented
- a 32-bit read/write register
- accessible in Secure privileged modes only
- a Restricted access register, meaning it exists only in the Secure state.

The format of the MVBAR is:

31 … 5	4 … 0
Monitor_Vector_Base_Address	(0) (0) (0) (0) (0)

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

VBAR のフォーマットは次のとおりです。

31 … 5	4 … 0
ベクタ ベースアドレス	(0) (0) (0) (0) (0)

VBAR のセキュアコピーには、セキュア例外ベースアドレスとして記述される、セキュア状態のベクタ ベースアドレスが保持されます。

VBAR の非セキュアコピーには、非セキュア例外ベースアドレスとして記述される、非セキュア状態のベクタ ベースアドレスが保持されます。

ベクタ ベースアドレス、ビット [31:5]

ノーマル例外ベクタのベースアドレスのビット [31:5]。例外ベクタのビット [4:0] が例外オフセットです。P. B1-31 の表 B1-3 を参照して下さい。

ビット [4:0] 予約、UNK/SBZP

例外アドレスを決定するための VBAR レジスタの使用方法の詳細については、*例外ベクタと例外ベースアドレス* P. B1-30 を参照して下さい。

VBAR へのアクセス

VBAR にアクセスするには、`<opc1>` を 0、`<CRn>` を c12、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c12,c0,0    ; Read  CP15 Vector Base Address Register
MCR p15,0,<Rt>,c12,c0,0    ; Write CP15 Vector Base Address Register
```

B3.12.41 c12、モニタベクタ ベースアドレス レジスタ (MVBAR)

モニタベクタ ベースアドレス レジスタ (MVBAR) では、モニタモードで処理されるすべての例外について、例外ベースアドレスが提供されます。*例外ベクタと例外ベースアドレス* P. B1-30 を参照して下さい。

MVBAR の特徴は次のとおりです。

- セキュリティ拡張機能が実装されている場合のみ存在します。
- 32 ビットの読み出し / 書き込みレジスタ
- セキュア特権モードでのみアクセス可能
- 限定アクセスレジスタで、セキュア状態でのみ存在します。

MVBAR のフォーマットは次のとおりです。

31 … 5	4 … 0
モニタベクタ ベースアドレス	(0) (0) (0) (0) (0)

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The reset value of the MVBAR is UNKNOWN. The MVBAR must be programmed as part of the boot sequence.

Monitor_Vector_Base_Address, bits [31:5]

Bits [31:5] of the base address of the exception vectors for exceptions that are handled in Monitor mode. Bits [4:0] of an exception vector is the exception offset, see Table B1-3 on page B1-31.

Bits [4:0] Reserved, UNK/SBZP.

For details of how the MVBAR is used to determine the exception addresses see *Exception vectors and the exception base address* on page B1-30.

Accessing the MVBAR

To access the MVBAR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c12, `<CRm>` set to c0, and `<opc2>` set to 1. For example:

```
MRC p15,0,<Rt>,c12,c0,1    ; Read  CP15 Monitor Vector Base Address Register
MCR p15,0,<Rt>,c12,c0,1    ; Write CP15 Monitor Vector Base Address Register
```

B3.12.42 c12, Interrupt Status Register (ISR)

The Interrupt Status Register, ISR, shows whether an IRQ, FIQ or external abort is pending.

The ISR is:

- present only when the Security Extensions are implemented
- a 32-bit read-only register
- accessible only in privileged modes.
- a Common register, meaning it is available in the Secure and Non-secure states.

The format of the ISR is:

31 … 9	8	7	6	5 … 0
UNK	A	I	F	(0) (0) (0) (0) (0) (0)

Bits [31:9] Reserved, UNK.

A, bit [8] External abort pending flag:

0 no pending external abort

1 an external abort is pending.

I, bit [7] Interrupt pending flag. Indicates whether an IRQ interrupt is pending:

0 no pending IRQ

1 an IRQ interrupt is pending.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

MVBAR のリセット時の値は未知です。MVBAR は、ブートシーケンスの一部としてプログラムする必要があります。

モニタベクタ ベースアドレス、ビット [31:5]

モニタモードで処理される例外についての、例外ベクタのベースアドレスのビット [31:5]。例外ベクタのビット [4:0] が例外オフセットです。P. B1-31 の表 B1-3 を参照して下さい。

ビット [4:0] 予約、UNK/SBZP

例外アドレスを決定するための MVBAR の使用方法の詳細については、*例外ベクタと例外ベースアドレス* P. B1-30 を参照して下さい。

MVBAR へのアクセス

MVBAR にアクセスするには、`<opc1>` を 0、`<CRn>` を c12、`<CRm>` を c0、`<opc2>` を 1 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c12,c0,1    ; Read  CP15 Monitor Vector Base Address Register
MCR p15,0,<Rt>,c12,c0,1    ; Write CP15 Monitor Vector Base Address Register
```

B3.12.42 c12、割り込みステータスレジスタ (ISR)

割り込みステータスレジスタ (ISR) は、IRQ、FIQ、または外部アボートが保留中かどうかを示します。

ISR の特徴は次のとおりです。

- セキュリティ拡張機能が実装されている場合のみ存在します。
- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- 共通レジスタで、セキュア状態と非セキュア状態で使用できます。

ISR のフォーマットは次のとおりです。

31 … 9	8	7	6	5 … 0
UNK	A	I	F	(0) (0) (0) (0) (0) (0)

ビット [31:9] 予約、UNK

A、ビット [8] 外部アボート保留フラグ

0 保留中の外部アボートはありません。

1 外部アボートが保留中です。

I、ビット [7] 割り込み保留フラグ。IRQ 割り込みが保留中かどうかを示します。

0 保留中の IRQ はありません。

1 IRQ 割り込みが保留中です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

F, bit [7] Fast interrupt pending flag. Indicates whether an FIQ fast interrupt is pending:

0 no pending FIQ

1 an FIQ fast interrupt is pending.

Bits [5:0] Reserved, UNK/SBZP.

The bit positions of the A, I and F flags in the ISR match the A, I and F flag bits in the CPSR, see *Program Status Registers (PSRs)* on page B1-14. This means the same masks can be used to extract the flags from the register value.

Note

- The ISR.F and ISR.I bits directly reflect the state of the FIQ and IRQ inputs.
- the ISR.A bit is set when an asynchronous abort is generated and is cleared automatically when the abort is taken.

Accessing the ISR

To access the ISR you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c12, `<CRm>` set to c1, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c12,c1,0    ; Read Interrupt Status Register
```

B3.12.43 CP15 c13, Process, context and thread ID registers

The CP15 c13 registers are used for:

- a Context ID register
- three software Thread ID registers
- an FCSE Process ID Register.

Note

From ARMv6, use of the FCSE is a deprecated, and in ARMv7 the FCSE is an optional component of a VMSA implementation. ARM expects the FCSE will become obsolete during the lifetime of ARMv7. However, every ARMv7-A implementation must include the FCSE Process ID Register.

Figure B3-25 on page B3-152 shows the CP15 c13 registers:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

F、ビット [7] 高速割り込み保留フラグ。FIQ 高速割り込みが保留中かどうかを示します。

0 保留中の FIQ はありません。

1 FIQ 高速割り込みが保留中です。

ビット [5:0] 予約、UNK/SBZP

ISR の A、I、F フラグのビット位置は、CPSR の A、I、F フラグのビットと一致します。*プログラムステータス レジスタ (PSR)* P. B1-14 を参照して下さい。これは、同じマスクを使用してレジスタの値からフラグを抽出できることを意味します。

注

- ISR.F ビットと ISR.I ビットには、FIQ および IRQ 入力の状態が直接反映されます。
- ISR.A ビットは、非同期アボートが生成され、そのアボートの取得時に自動的にクリアされた場合にセットされます。

ISR へのアクセス

ISR にアクセスするには、`<opc1>` を 0、`<CRn>` を c12、`<CRm>` を c1、`<opc2>` を 0 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,0,<Rt>,c12,c1,0    ; Read Interrupt Status Register
```

B3.12.43 CP15 c13, Process, context and thread ID registers

CP15 c13 レジスタの用途は次のとおりです。

- コンテキスト ID レジスタ
- 3 つのソフトウェアスレッド ID レジスタ
- FCSE プロセス ID レジスタ

注

ARMv6 およびそれ以降では FCSE の使用は非推奨で、ARMv7 では FCSE は VMSA 実装のオプションコンポーネントです。ARM は、ARMv7 の存続期間中に FCSE の廃止を予定しています。ただし、すべての ARMv7-A 実装には、FCSE プロセス ID レジスタを含める必要があります。

CP15 c13 レジスタを、P. B3-152 の図 B3-25 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure B3-25 CP15 c13 registers in a VMSA implementation

CP15 c13 encodings not shown in Figure B3-25 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B3-69.

The CP15 c13 registers are described in:

- *c13, FCSE Process ID Register (FCSEIDR)*
- *c13, Context ID Register (CONTEXTIDR)* on page B3-153
- *CP15 c13 Software Thread ID registers* on page B3-154.

B3.12.44 c13, FCSE Process ID Register (FCSEIDR)

The FCSE Process ID Register, FCSEIDR, identifies the current *Process ID* (PID) for the *Fast Context Switch Extension* (FCSE). In ARMv7, the FCSE is optional. However, the FCSEIDR must be implemented regardless of whether the FCSE is implemented. Software can access this register to determine whether the FCSE is implemented.

The FCSEIDR:

- Is a 32-bit register, with access that depends on whether the FCSE is implemented:

  FCSE implemented: . the register is read/write

  FCSE not implemented: . the register is RAZ/WI.

- Is accessible only in privileged modes.
- When implemented as a read/write register, has a defined reset value of 0. When the Security Extensions are implemented, this reset value applies only to the Secure copy of the register, and software must program the Non-secure copy of the register with the required value.
- When the Security Extensions are implemented, is a Banked register.

  When the Security Extensions are implemented and the FCSE is implemented, write access to the Secure copy of the FCSEIDR is disabled when the **CP15SDISABLE** signal is asserted HIGH.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

図 B3- 25 VMSA 実装の CP15 c13 レジスタ

図 B3- 25 に示されていない CP15 c13 エンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B3- 69 を参照して下さい。

CP15 c13 レジスタについては、次のサブセクションで説明します。

- *c13、FCSE プロセス ID レジスタ (FCSEIDR)*
- *c13、コンテキスト ID レジスタ (CONTEXTIDR)* P. B3- 153
- *CP15 c13、ソフトウェアスレッド ID レジスタ* P. B3- 154

B3.12.44 c13、FCSE プロセス ID レジスタ (FCSEIDR)

FCSE プロセス ID レジスタ (FCSEIDR) は、*高速コンテキストスイッチ拡張機能* (FCSE) の現在の *プロセス ID*(PID) を識別します。ARMv7 では、FCSE はオプションです。ただし、FCSE が実装されているかどうかにかかわらず、FCSEIDR を実装する必要があります。ソフトウェアは、このレジスタにアクセスして、FCSE が実装されているかどうかを判定できます。

FCSEIDR の特徴は次のとおりです。

- 32 ビットレジスタで、FCSE が実装されているかどうかによってアクセスが異なります。

  FCSE が実装されている場合：レジスタは読み出し / 書き込み可能です。

  FCSE が実装されていない場合：レジスタは RAZ/WI です。

- 特権モードでのみアクセス可能
- 読み出し / 書き込みレジスタとして実装されている場合、リセット時の値は 0 に定義されます。セキュリティ拡張機能が実装されている場合、このリセット時の値はレジスタのセキュアコピーにのみ適用されるため、ソフトウェアでレジスタの非セキュアコピーを必要な値にプログラムする必要があります。
- セキュリティ拡張機能が実装されている場合、バンクレジスタです。

  セキュリティ拡張機能が実装され、FCSE が実装されている場合、FCSEIDR のセキュアコピーへの書き込みアクセスは、**CP15SDISABLE** 信号が HIGH にアサートされているときには不可能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the FCSEIDR is:

31 – 25	24 – 0
PID	UNK/SBZP

PID, bits [31:25]

The current Process ID, for the FCSE. If the FCSE is not implemented this field is RAZ/WI.

Bits [24:0] Reserved. If the FCSE is not implemented this field is RAZ/WI.

If the FCSE is implemented, the value of this field is UNKNOWN on reads and Should-Be-Zero-or-Preserved on writes.

Note

- When the PID is written, the overall virtual-to-physical address mapping changes. Because of this, you must ensure that instructions that might have been prefetched already are not affected by the address mapping change.
- From ARMv6, use of the FCSE is deprecated, and in ARMv7 the FCSE is optional.

Accessing the FCSEIDR

To access the FCSEIDR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c13, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c13,c0,0    ; Read  CP15 FCSE PID Register
MCR p15,0,<Rt>,c13,c0,0    ; Write CP15 FCSE PID Register
```

B3.12.45 c13, Context ID Register (CONTEXTIDR)

The Context ID Register, CONTEXTIDR, identifies the current:

- *Process Identifier* (PROCID)
- *Address Space Identifier* (ASID).

The value of the whole of this register is called the *Context ID* and is used by:

- the debug logic, for Linked and Unlinked Context ID matching, see *Breakpoint debug events* on page C3-5 and *Watchpoint debug events* on page C3-15.
- the trace logic, to identify the current process.

The ASID field value is used by many memory management functions.

The CONTEXTIDR is:

- a 32-bit read/write register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Banked register.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

FCSEIDR のフォーマットは次のとおりです。

31 – 25	24 – 0
PID	UNK/SBZP

PID、ビット [31:25]

FCSE の現在のプロセス ID。FCSE が実装されていない場合、このフィールドは RAZ/WI です。

ビット [24:0] 予約。FCSE が実装されていない場合、このフィールドは RAZ/WI です。

FCSE が実装されている場合、このフィールドの読み出しの値は未知で、書き込みは常に 0 または保持です。

注

- PID が書き込まれると、仮想アドレスから物理アドレスへのマッピング全体が変更されます。そのため、すでにプリフェッチされている可能性のある命令が、アドレスマッピングの変更の影響を受けないことを保証する必要があります。
- ARMv6 およびそれ以降では FCSE の使用は非推奨で、ARMv7 では FCSE がオプションです。

FCSEIDR へのアクセス

FCSEIDR にアクセスするには、`<opc1>` を 0、`<CRn>` を c13、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c13,c0,0    ; Read  CP15 FCSE PID Register
MCR p15,0,<Rt>,c13,c0,0    ; Write CP15 FCSE PID Register
```

B3.12.45 c13、コンテキスト ID レジスタ (CONTEXTIDR)

コンテキスト ID レジスタ (CONTEXTIDR) は、次の識別子について現在の値を識別します。

- *プロセス識別子*(PROCID)
- *アドレス空間識別子*(ASID)

このレジスタ全体の値は*コンテキスト ID* と呼ばれ、次の部分で使用されます。

- リンク付きおよびリンクなしのコンテキスト ID 一致用のデバッグロジック（*ブレークポイントデバッグ イベント* P. C3-5 と*ウォッチポイントデバッグ イベント* P. C3-15 を参照）
- トレースロジックで、現在のプロセスを識別するため

ASID フィールドの値は、多くのメモリ管理機能で使用されます。

CONTEXTIDR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the CONTEXTIDR is:

31 … 8	7 … 0
PROCID	ASID

PROCID, bits [31:8]

Process Identifier. This field must be programmed with a unique value that identifies the current process. It is used by the trace logic and the debug logic to identify the process that is running currently.

ASID, bits [7:0]

Address Space Identifier. This field is programmed with the value of the current ASID.

Using the CONTEXTIDR

For information about the synchronization of changes to the CONTEXTIDR see *Changes to CP15 registers and the memory order model* on page B3-77. There are particular synchronization requirements when changing the ASID and Translation Table Base Registers, see *Synchronization of changes of ASID and TTBR* on page B3-60.

Accessing the CONTEXTIDR

To access the CONTEXTIDR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c13, `<CRm>` set to c0, and `<opc2>` set to 1. For example:

```
MRC p15,0,<Rt>,c13,c0,1    ; Read  CP15 Context ID Register
MCR p15,0,<Rt>,c13,c0,1    ; Write CP15 Context ID Register
```

B3.12.46 CP15 c13 Software Thread ID registers

The Software Thread ID registers provide locations where software can store thread identifying information, for OS management purposes. These registers are never updated by the hardware.

The Software Thread ID registers are:

- three 32-bit register read/write registers:
  - User Read/Write Thread ID Register, TPIDRURW
  - User Read-only Thread ID Register, TPIDRURO
  - Privileged Only Thread ID Register, TPIDRPRW.
- accessible in different modes:
  - the User Read/Write Thread ID Register is read/write in unprivileged and privileged modes
  - the User Read-only Thread ID Register is read-only in User mode, and read/write in privileged modes
  - the Privileged Only Thread ID Register is only accessible in privileged modes, and is read/write.
- when the Security Extensions are implemented, Banked registers
- introduced in ARMv7.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- セキュリティ拡張機能が実装されている場合、バンクレジスタです。

CONTEXTIDR のフォーマットは次のとおりです。

31 … 8	7 … 0
PROCID	ASID

PROCID、ビット [31:8]

プロセス識別子。このフィールドは、現在のプロセスを識別する一意の値にプログラムする必要があります。この値は、トレースロジックとデバッグロジックで、実行中のプロセスを識別するために使用されます。

ASID、ビット [7:0]

アドレス空間識別子。このフィールドは、現在の ASID の値にプログラムされます。

CONTEXTIDR の使用

CONTEXTIDR の変更の同期方法については、*CP15 レジスタの変更とメモリオーダ モデル* P. B3-77 を参照して下さい。ASID と変換テーブル ベースレジスタを変更する場合は、特別な同期化要件があります。*ASID と TTBR の変更の同期化* P. B3-60 を参照して下さい。

CONTEXTIDR へのアクセス

CONTEXTIDR にアクセスするには、`<opc1>` を 0、`<CRn>` を c13、`<CRm>` を c0、`<opc2>` を 1 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c13,c0,1    ; Read  CP15 Context ID Register
MCR p15,0,<Rt>,c13,c0,1    ; Write CP15 Context ID Register
```

B3.12.46 CP15 c13、ソフトウェアスレッド ID レジスタ

ソフトウェアスレッド ID レジスタは、OS で管理するためのスレッド識別情報を、ソフトウェアで保存するための場所を提供します。これらのレジスタがハードウェアで更新されることはありません。

ソフトウェアスレッド ID レジスタの特徴は次のとおりです。

- 3 つの 32 ビットの読み出し / 書き込みレジスタ
  - ユーザ読み出し / 書き込みスレッド ID レジスタ、TPIDRURW
  - ユーザ読み出し専用スレッド ID レジスタ、TPIDRURO
  - 特権専用スレッド ID レジスタ、TPIDRPRW
- さまざまなモードでアクセス可能です。
  - ユーザ読み出し / 書き込みスレッド ID レジスタは、非特権モードと特権モードで読み出し / 書き込み可能です。
  - ユーザ読み出し専用スレッド ID レジスタは、ユーザモードで読み取り専用、特権モードで読み出し / 書き込み可能です。
  - 特権専用スレッド ID レジスタは、特権モードでのみアクセス可能で、読み出し / 書き込み可能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the Software Thread ID registers

To access the Software Thread ID registers you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c13, `<CRm>` set to c0, and `<opc2>` set to:

- 2 for the User Read/Write Thread ID Register, TPIDRURW
- 3 for the User Read-only Thread ID Register, TPIDRURO
- 4 for the Privileged Only Thread ID Register, TPIDRPRW.

For example:

```
MRC p15, 0, <Rt>, c13, c0, 2 ; Read  CP15 User Read/Write Thread ID Register
MCR p15, 0, <Rt>, c13, c0, 2 ; Write CP15 User Read/Write Thread ID Register
MRC p15, 0, <Rt>, c13, c0, 3 ; Read  CP15 User Read-only  Thread ID Register
MCR p15, 0, <Rt>, c13, c0, 3 ; Write CP15 User Read-only  Thread ID Register
MRC p15, 0, <Rt>, c13, c0, 4 ; Read  CP15 Privileged Only Thread ID Register
MCR p15, 0, <Rt>, c13, c0, 4 ; Write CP15 Privileged Only Thread ID Register
```

B3.12.47 CP15 c14, Not used

CP15 c14 is not used on any ARMv7 implementation, see *Unallocated CP15 encodings* on page B3-69.

B3.12.48 CP15 c15, IMPLEMENTATION DEFINED registers

CP15 c15 is reserved for IMPLEMENTATION DEFINED purposes. ARMv7 does not impose any restrictions on the use of the CP15 c15 encodings. The documentation of the ARMv7 implementation must describe fully any registers implemented in CP15 c15. Normally, for processor implementations by ARM, this information is included in the *Technical Reference Manual* for the processor.

Typically, CP15 c15 is used to provide test features, and any required configuration options that are not covered by this manual.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- セキュリティ拡張機能が実装されている場合、バンクレジスタです。
- ARMv7 で導入されました。

ソフトウェアスレッド ID レジスタへのアクセス

ソフトウェアスレッド ID レジスタにアクセスするには、`<opc1>` を 0、`<CRn>` を c13、`<CRm>` を c0、`<opc2>` を次の値に設定して、CP15 レジスタを読み出し / 書き込みします。

- ユーザ読み出し / 書き込みスレッド ID レジスタ (TPIDRURW) の場合は 2
- ユーザ読み出し専用スレッド ID レジスタ (TPIDRURO) の場合は 3
- 特権専用スレッド ID レジスタ (TPIDRPRW) の場合は 4

次に例を示します。

```
MRC p15, 0, <Rt>, c13, c0, 2 ; Read  CP15 User Read/Write Thread ID Register
MCR p15, 0, <Rt>, c13, c0, 2 ; Write CP15 User Read/Write Thread ID Register
MRC p15, 0, <Rt>, c13, c0, 3 ; Read  CP15 User Read-only  Thread ID Register
MCR p15, 0, <Rt>, c13, c0, 3 ; Write CP15 User Read-only  Thread ID Register
MRC p15, 0, <Rt>, c13, c0, 4 ; Read  CP15 Privileged Only Thread ID Register
MCR p15, 0, <Rt>, c13, c0, 4 ; Write CP15 Privileged Only Thread ID Register
```

B3.12.47 CP15 c14、未使用

ARMv7 実装では CP15 c14 が使用されません。*未割り当ての CP15 エンコード* P. B3- 69 を参照して下さい。

B3.12.48 CP15 c15、実装定義のレジスタ

CP15 c15 は実装定義の目的に予約されています。ARMv7 では、CP15 c15 エンコードの使用に関する制限はありません。ARMv7 実装のドキュメントには、CP15 c15 に実装されるすべてのレジスタを完全に記載する必要があります。通常、ARM によるプロセッサ実装の場合は、この情報がプロセッサの *テクニカルリファレンス マニュアル*に含まれています。

多くの場合、CP15 c15 は、テスト機能と、本書でカバーされていない必要な構成オプションを提供するために使用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.13 Pseudocode details of VMSA memory system operations

This section contains pseudocode describing VMSA memory operations. The following subsections describe the pseudocode functions:

- *Alignment fault*
- *FCSE translation*
- *Address translation* on page B3-157
- *Domain checking* on page B3-157
- *TLB operations* on page B3-158
- *Translation table walk* on page B3-158.

See also the pseudocode for general memory system operations in *Pseudocode details of general memory system operations* on page B2-29.

B3.13.1 Alignment fault

The following pseudocode describes the generation of an Alignment fault Data Abort exception:

```
// AlignmentFaultV()
// =================

AlignmentFaultV(bits(32) address, boolean iswrite)

    mva = FCSETranslate(address);
    DataAbort(mva, bits(4) UNKNOWN, boolean UNKNOWN, iswrite, DAbort_Alignment);
```

B3.13.2 FCSE translation

The following pseudocode describes the FCSE translation:

```
// FCSETranslate()
// ===============

bits(32) FCSETranslate(bits(32) va)
    if va<31:25> == '0000000' then
        mva = FCSEIDR.PID : va<24:0>;
    else
        mva = va;
    return mva;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.13 VMSA メモリシステム操作の擬似コードの詳細

このセクションでは、VMSA メモリ操作の擬似コードを紹介します。次のサブセクションで、擬似コード関数について説明します。

- *アライメントフォールト*
- *FCSE 変換*
- *アドレス変換* P. B3-157
- *ドメインチェック* P. B3-157
- *TLB 操作* P. B3-158
- *変換テーブルウォーク* P. B3-158

一般的なメモリシステム操作の擬似コードの詳細 P. B2-29 にある、一般的なメモリシステム操作の擬似コードも参照して下さい。

B3.13.1 アライメントフォールト

次の擬似コードは、アライメントフォールト データアボート例外の生成を示しています。

```
// AlignmentFaultV()
// =================

AlignmentFaultV(bits(32) address, boolean iswrite)

    mva = FCSETranslate(address);
    DataAbort(mva, bits(4) UNKNOWN, boolean UNKNOWN, iswrite, DAbort_Alignment);
```

B3.13.2 FCSE 変換

次の擬似コードは、FCSE 変換を示しています。

```
// FCSETranslate()
// ===============

bits(32) FCSETranslate(bits(32) va)
    if va<31:25> == '0000000' then
        mva = FCSEIDR.PID : va<24:0>;
    else
        mva = va;
    return mva;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.13.3 Address translation

The following pseudocode describes address translation in a VMSA implementation:

```
// TranslateAddressV()
// ===================

AddressDescriptor TranslateAddressV(bits(32) va, boolean ispriv, boolean iswrite)

    mva = FCSETranslate(va);

    if SCTLR.M == '1' then  // MMU is enabled
        (tlbhit, tlbrecord) = CheckTLB(CONTEXTIDR.ASID, mva);
        if !tlbhit then
            tlbrecord = TranslationTableWalk(mva, iswrite);
        if CheckDomain(tlbrecord.domain, mva, tlbrecord.sectionnotpage, iswrite) then
            CheckPermission(tlbrecord.perms, mva, tlbrecord.sectionnotpage, iswrite, ispriv);
    else
        tlbrecord = TranslationTableWalk(mva, iswrite);

    return tlbrecord.addrdesc;
```

B3.13.4 Domain checking

The following pseudocode describes domain checking:

```
// CheckDomain()
// =============

boolean CheckDomain(bits(4) domain, bits(32) mva, boolean sectionnotpage, boolean iswrite)

    bitpos = 2*UInt(domain);
    case DACR<bitpos+1:bitpos> of
        when '00'  DataAbort(mva, domain, sectionnotpage, iswrite, DAbort_Domain);
        when '01'  permissioncheck = TRUE;
        when '10'  UNPREDICTABLE;
        when '11'  permissioncheck = FALSE;

    return permissioncheck;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.13.3 アドレス変換

次の擬似コードは、VMSA 実装でのアドレス変換を示しています。


```
// TranslateAddressV()
// ===================

AddressDescriptor TranslateAddressV(bits(32) va, boolean ispriv, boolean iswrite)

    mva = FCSETranslate(va);

    if SCTLR.M == '1' then  // MMU is enabled
        (tlbhit, tlbrecord) = CheckTLB(CONTEXTIDR.ASID, mva);
        if !tlbhit then
            tlbrecord = TranslationTableWalk(mva, iswrite);
        if CheckDomain(tlbrecord.domain, mva, tlbrecord.sectionnotpage, iswrite) then
            CheckPermission(tlbrecord.perms, mva, tlbrecord.sectionnotpage, iswrite, ispriv);
    else
        tlbrecord = TranslationTableWalk(mva, iswrite);

    return tlbrecord.addrdesc;
```


B3.13.4 ドメインチェック

次の擬似コードは、ドメインチェックを示しています。


```
// CheckDomain()
// =============

boolean CheckDomain(bits(4) domain, bits(32) mva, boolean sectionnotpage, boolean iswrite)

    bitpos = 2*UInt(domain);
    case DACR<bitpos+1:bitpos> of
        when '00'  DataAbort(mva, domain, sectionnotpage, iswrite, DAbort_Domain);
        when '01'  permissioncheck = TRUE;
        when '10'  UNPREDICTABLE;
        when '11'  permissioncheck = FALSE;

    return permissioncheck;
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B3.13.5 TLB operations

The TLBRecord type represents the contentsof a TLB entry:

```
// Types of TLB entry

enumeration TLBRecType = { TLBRecType_SmallPage,
                           TLBRecType_LargePage,
                           TLBRecType_Section,
                           TLBRecType_Supersection,
                           TLBRecType_MMUDisabled
                         };

type TLBRecord is (
    Permissions       perms,
    bit               nG,             // '0' = Global, '1' = not Global
    bits(4)           domain,
    boolean           sectionnotpage,
    TLBRecType        type,
    AddressDescriptor addrdesc
)
```

The CheckTLB() function checks whether the TLB contains an entry that matches an ASID and address, and returns TRUE and the matching TLBRecord if so. Otherwise, it returns FALSE and an UNKNOWN TLBRecord.

```
(boolean, TLBRecord) CheckTLB(bits(8) asid, bits(32) address)
```

The AssignToTLB() procedure supplies an ASID and new TLBRecord to the TLB, for possible allocation to a TLB entry. It is IMPLEMENTATION DEFINED under what circumstances this allocation takes place, and TLB entries might also be allocated at other times.

```
AssignToTLB(bits(8) asid, bits(32) mva, TLBRecord entry)
```

B3.13.6 Translation table walk

The following pseudocode describes the translation table walk operation:

```
// TranslationTableWalk()
// ======================
//
// Returns a result of a translation table walk in TLBRecord form.

TLBRecord TranslationTableWalk(bits(32) mva, boolean is_write)

    TLBRecord         result;
    AddressDescriptor l1descaddr;
    AddressDescriptor l2descaddr;

    if SCTLR.M == '1' then  // MMU is enabled

        domain = bits(4) UNKNOWN;  // For Data Abort exceptions found before a domain is known

        // Determine correct Translation Table Base Register to use.
        n = UInt(TTBCR.N);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B3.13.5 TLB 操作

TLBRecord タイプは TLB エントリの内容を表します。

```
// Types of TLB entry

enumeration TLBRecType = { TLBRecType_SmallPage,
                           TLBRecType_LargePage,
                           TLBRecType_Section,
                           TLBRecType_Supersection,
                           TLBRecType_MMUDisabled
                         };

type TLBRecord is (
    Permissions       perms,
    bit               nG,             // '0' = Global, '1' = not Global
    bits(4)           domain,
    boolean           sectionnotpage,
    TLBRecType        type,
    AddressDescriptor addrdesc
)
```

CheckTLB() 関数は、TLB に ASID およびアドレスと一致するエントリが含まれているかどうかをチェックして、含まれていれば、TRUE および一致する TLBRecord を返します。そうでない場合は、FALSE および未知の TLBRecord を返します。

```
(boolean, TLBRecord) CheckTLB(bits(8) asid, bits(32) address)
```

AssignToTLB() プロシージャは、ASID および新しい TLBRecord を、TLB エントリへの可能な割り当てとして TLB に供給します。この割り当てがどの状況で行われるかは実装定義で、別の状況でも TLB エントリが割り当てられる場合があります。

```
AssignToTLB(bits(8) asid, bits(32) mva, TLBRecord entry)
```

B3.13.6 変換テーブルウォーク

次の擬似コードは、変換テーブルウォーク操作を示しています。

```
// TranslationTableWalk()
// ======================
//
// Returns a result of a translation table walk in TLBRecord form.

TLBRecord TranslationTableWalk(bits(32) mva, boolean is_write)

    TLBRecord         result;
    AddressDescriptor l1descaddr;
    AddressDescriptor l2descaddr;

    if SCTLR.M == '1' then  // MMU is enabled

        domain = bits(4) UNKNOWN;  // For Data Abort exceptions found before a domain is known

        // Determine correct Translation Table Base Register to use.
        n = UInt(TTBCR.N);
        if n == 0 || IsZero(mva<31:(32-n)>) then
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        if n == 0 || IsZero(mva<31:(32-n)>) then
            ttbr = TTBR0;
            disabled = (TTBCR.PD0 == '1');
        else
            ttbr = TTBR1;
            disabled = (TTBCR.PD1 == '1');
            n = 0;  // TTBR1 translation always works like N=0 TTBR0 translation

        // Check this Translation Table Base Register is not disabled.
        if HaveSecurityExt() && disabled == '1' then
            DataAbort(mva, domain, TRUE, is_write, DAbort_Translation);

        // Obtain level 1 descriptor.
        l1descaddr.paddress.physicaladdress = ttbr<31:(14-n)> : mva<(31-n):20> : '00';
        l1descaddr.paddress.physicaladdressext = '00000000';
        l1descaddr.paddress.NS = if IsSecure() then '0' else '1';
        l1descaddr.memattrs.type = MemType_Normal;
        l1descaddr.memattrs.shareable = (ttbr<1> == '1');
        l1descaddr.memattrs.outershareable = (ttbr<5> == '0') &&  (ttbr<1> == '1');
        l1descaddr.memattrs.outerattrs = ttbr<4:3>;

        if HaveMPExt() then
            1descaddr.memattrs.innerattrs = ttbr<0>:ttbr<6>;
        else
            if ttbr<0> == '0' then
                l1descaddr.memattrs.innerattrs = '00';
            else
                IMPLEMENTATION_DEFINED set l1descaddr.memattrs.innerattrs to one of
'01','10','11';
        l1desc = _Mem[l1descaddr,4];

        // Process level 1 descriptor.
        case l1desc<1:0> of
            when '00', '11'  // Fault, Reserved
                DataAbort(mva, domain, TRUE, is_write, DAbort_Translation);

            when '01'        // Section or Supersection
                texcb = l1desc<14:12,3,2>;
                S = l1desc<16>;
                ap = l1desc<15,11:10>;
                xn = l1desc<4>;
                nG = l1desc<17>;
                sectionnotpage = TRUE;
                NS = l1desc<19>;

                if SCTLR.AFE == '1' && l1desc<10> == '0' then
                    if SCTLR.HA == '0' then
                        DataAbort(mva, domain, sectionnotpage, is_write, DAbort_AccessFlag);
                    else  // Hardware-managed access flag must be set in memory
                        _Mem[l1descaddr,4]<10> = '1';

                if l1desc<18> == '0' then  // Section
                    domain = l1desc<8:5>;
                    type = TLBRecType_Section;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
                    ttbr = TTBR0;
                    disabled = (TTBCR.PD0 == '1');
            else
                    ttbr = TTBR1;
                    disabled = (TTBCR.PD1 == '1');
                    n = 0;  // TTBR1 translation always works like N=0 TTBR0 translation

            // Check this Translation Table Base Register is not disabled.
            if HaveSecurityExt() && disabled == '1' then
                DataAbort(mva, domain, TRUE, is_write, DAbort_Translation);

            // Obtain level 1 descriptor.
            l1descaddr.paddress.physicaladdress = ttbr<31:(14-n)> : mva<(31-n):20> : '00';
            l1descaddr.paddress.physicaladdressext = '00000000';
            l1descaddr.paddress.NS = if IsSecure() then '0' else '1';
            l1descaddr.memattrs.type = MemType_Normal;
            l1descaddr.memattrs.shareable = (ttbr<1> == '1');
            l1descaddr.memattrs.outershareable = (ttbr<5> == '0') &&  (ttbr<1> == '1');
            l1descaddr.memattrs.outerattrs = ttbr<4:3>;

            if HaveMPExt() then
                1descaddr.memattrs.innerattrs = ttbr<0>:ttbr<6>;
            else
                if ttbr<0> == '0' then
                    l1descaddr.memattrs.innerattrs = '00';
                else
                    IMPLEMENTATION_DEFINED set l1descaddr.memattrs.innerattrs to one of
'01','10','11';
            l1desc = _Mem[l1descaddr,4];

            // Process level 1 descriptor.
            case l1desc<1:0> of
                when '00', '11'  // Fault, Reserved
                    DataAbort(mva, domain, TRUE, is_write, DAbort_Translation);

                when '01'        // Section or Supersection
                    texcb = l1desc<14:12,3,2>;
                    S = l1desc<16>;
                    ap = l1desc<15,11:10>;
                    xn = l1desc<4>;
                    nG = l1desc<17>;
                    sectionnotpage = TRUE;
                    NS = l1desc<19>;

                    if SCTLR.AFE == '1' && l1desc<10> == '0' then
                        if SCTLR.HA == '0' then
                            DataAbort(mva, domain, sectionnotpage, is_write, DAbort_AccessFlag);
                        else  // Hardware-managed access flag must be set in memory
                            _Mem[l1descaddr,4]<10> = '1';

                    if l1desc<18> == '0' then  // Section
                        domain = l1desc<8:5>;
                        type = TLBRecType_Section;
                        physicaladdressext = '00000000';
                        physicaladdress = l1desc<31:20> : mva<19:0>;
                    else                        // Supersection
                        domain = '0000';
                        type = TLBRecType_Supersection;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
                    physicaladdressext = '00000000';
                    physicaladdress = l1desc<31:20> : mva<19:0>;
                else                        // Supersection
                    domain = '0000';
                    type = TLBRecType_Supersection;
                    physicaladdressext = l1desc<8:5,23:20>;
                    physicaladdress = l1desc<31:24> : mva<23:0>;

            when '10'        // Large page or Small page
                domain = l1desc<8:5>;
                sectionnotpage = FALSE;
                NS = l1desc<3>;

                // Obtain level 2 descriptor.
                l2descaddr.paddress.physicaladdress = l1desc<31:10> : mva<19:12> : '00';
                l2descaddr.paddress.physicaladdressext = '00000000';
                l2descaddr.paddress.NS = if IsSecure() then '0' else '1';
                l2descaddr.memattrs = l1descaddr.memattrs;
                l2desc = _Mem[l2descaddr,4];

                // Process level 2 descriptor.
                if l2desc<1:0> == '00' then
                    DataAbort(mva, domain, sectionnotpage, is_write, DAbort_Translation);

                S = l2desc<10>;
                ap = l2desc<9,5:4>;
                nG = l2desc<11>;

                if SCTLR.AFE == '1' && l2desc<4> == '0' then
                    if SCTLR.HA == '0' then
                        DataAbort(mva, domain, sectionnotpage, is_write, DAbort_AccessFlag);
                    else  // Hardware-managed access flag must be set in memory
                        _Mem[l2descaddr,4]<4> = '1';

                if l2desc<1> == '0' then   // Large page
                    texcb = l2desc<14:12,3,2>
                    xn = l2desc<15>;
                    type = TLBRecType_LargePage;
                    physicaladdressext = '00000000';
                    physicaladdress = l2desc<31:16> : mva<15:0>;
                else                        // Small page
                    texcb = l2desc<8:6,3,2>;
                    xn = l2desc<0>;
                    type = TLBRecType_SmallPage;
                    physicaladdressext = '00000000';
                    physicaladdress = l2desc<31:12> : mva<11:0>;

        else  // MMU is disabled

            texcb = '00000';
            S = '1';
            ap = bits(3) UNKNOWN;
            xn = bit UNKNOWN;
            nG = bit UNKNOWN;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
                    physicaladdressext = l1desc<8:5,23:20>;
                    physicaladdress = l1desc<31:24> : mva<23:0>;

            when '10'        // Large page or Small page
                domain = l1desc<8:5>;
                sectionnotpage = FALSE;
                NS = l1desc<3>;

                // Obtain level 2 descriptor.
                l2descaddr.paddress.physicaladdress = l1desc<31:10> : mva<19:12> : '00';
                l2descaddr.paddress.physicaladdressext = '00000000';
                l2descaddr.paddress.NS = if IsSecure() then '0' else '1';
                l2descaddr.memattrs = l1descaddr.memattrs;
                l2desc = _Mem[l2descaddr,4];

                // Process level 2 descriptor.
                if l2desc<1:0> == '00' then
                    DataAbort(mva, domain, sectionnotpage, is_write, DAbort_Translation);

                S = l2desc<10>;
                ap = l2desc<9,5:4>;
                nG = l2desc<11>;

                if SCTLR.AFE == '1' && l2desc<4> == '0' then
                    if SCTLR.HA == '0' then
                        DataAbort(mva, domain, sectionnotpage, is_write, DAbort_AccessFlag);
                    else  // Hardware-managed access flag must be set in memory
                        _Mem[l2descaddr,4]<4> = '1';

                if l2desc<1> == '0' then   // Large page
                    texcb = l2desc<14:12,3,2>
                    xn = l2desc<15>;
                    type = TLBRecType_LargePage;
                    physicaladdressext = '00000000';
                    physicaladdress = l2desc<31:16> : mva<15:0>;
                else                        // Small page
                    texcb = l2desc<8:6,3,2>;
                    xn = l2desc<0>;
                    type = TLBRecType_SmallPage;
                    physicaladdressext = '00000000';
                    physicaladdress = l2desc<31:12> : mva<11:0>;

        else  // MMU is disabled

            texcb = '00000';
            S = '1';
            ap = bits(3) UNKNOWN;
            xn = bit UNKNOWN;
            nG = bit UNKNOWN;
            domain = bits(4) UNKNOWN;
            sectionnotpage = boolean UNKNOWN;
            type = TLBRecType_MMUDisabled;
            physicaladdress = mva;
            physicaladdressext = '00000000';
            NS = if IsSecure() then '0' else '1';

        // Decode the TEX, C, B and S bits to produce the TLBRecord's memory attributes.
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    domain = bits(4) UNKNOWN;
    sectionnotpage = boolean UNKNOWN;
    type = TLBRecType_MMUDisabled;
    physicaladdress = mva;
    physicaladdressext = '00000000';
    NS = if IsSecure() then '0' else '1';

// Decode the TEX, C, B and S bits to produce the TLBRecord's memory attributes.

if SCTLR.TRE == '0' then
    if RemapRegsHaveResetValues() then
        result.addrdesc.memattrs = DefaultTEXDecode(texcb, S);
    else
        IMPLEMENTATION_DEFINED setting of result.addrdesc.memattrs;
else
    if SCTLR.M == '0' then
        result.addrdesc.memattrs = DefaultTEXDecode(texcb, S);
    else
        result.addrdesc.memattrs = RemappedTEXDecode(texcb, S);

// Set the rest of the TLBRecord, try to add it to the TLB, and return it.
result.perms.ap = ap;
result.perms.xn = xn;
result.nG = nG;
result.domain = domain;
result.sectionnotpage = sectionnotpage;
result.type = type;
result.addrdesc.paddress.physicaladdress = physicaladdress;
result.addrdesc.paddress.physicaladdressext = physicaladdressext;
result.addrdesc.paddress.NS = NS;

AssignToTLB(CONTEXTIDR.ASID, mva, result);

return result;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
if SCTLR.TRE == '0' then
    if RemapRegsHaveResetValues() then
        result.addrdesc.memattrs = DefaultTEXDecode(texcb, S);
    else
        IMPLEMENTATION_DEFINED setting of result.addrdesc.memattrs;
else
    if SCTLR.M == '0' then
        result.addrdesc.memattrs = DefaultTEXDecode(texcb, S);
    else
        result.addrdesc.memattrs = RemappedTEXDecode(texcb, S);

// Set the rest of the TLBRecord, try to add it to the TLB, and return it.
result.perms.ap = ap;
result.perms.xn = xn;
result.nG = nG;
result.domain = domain;
result.sectionnotpage = sectionnotpage;
result.type = type;
result.addrdesc.paddress.physicaladdress = physicaladdress;
result.addrdesc.paddress.physicaladdressext = physicaladdressext;
result.addrdesc.paddress.NS = NS;

AssignToTLB(CONTEXTIDR.ASID, mva, result);

return result;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter B4
Protected Memory System Architecture (PMSA)

This chapter provides a system-level view of the memory system. It contains the following sections:

- *About the PMSA* on page B4-2
- *Memory access control* on page B4-9
- *Memory region attributes* on page B4-11
- *PMSA memory aborts* on page B4-13
- *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18
- *CP15 registers for a PMSA implementation* on page B4-22
- *Pseudocode details of PMSA memory system operations* on page B4-79.

Note

For an ARMv7-R implementation, this chapter must be read with Chapter B2 *Common Memory System Architecture Features*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 B4 章
保護メモリシステム アーキテクチャ (PMSA)

本章では、システムレベルから見たメモリシステムについて説明します。本章は次のセクションから構成されています。

- *PMSA について* P. B4-2
- *メモリアクセスの制御* P. B4-9
- *メモリ領域の属性* P. B4-11
- *PMSA メモリアボート* P. B4-13
- *PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4-18
- *PMSA 実装の CP15 レジスタ* P. B4-22
- *PMSA メモリシステム操作の擬似コードの詳細* P. B4-79

注

ARMv7-R 実装の場合は、本章と第 B2 章*「一般的なメモリシステム アーキテクチャの機能」*をあわせて読んで下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.1 About the PMSA

The PMSA is based on a *Memory Protection Unit* (MPU). The PMSA provides a much simpler memory protection scheme than the MMU based VMSA described in Chapter B3 *Virtual Memory System Architecture (VMSA)*. The simplification applies to both the hardware and the software. A PMSAv7 processor is identified by the presence of the MPU Type Register, see *c0, MPU Type Register (MPUIR)* on page B4-36.

The main simplification is that the MPU does not use translation tables. Instead, System Control Coprocessor (CP15) registers are used to define *protection regions*. The protection regions eliminate the need for:

- hardware to perform translation table walks
- software to set up and maintain the translation tables.

The use of protection regions has the benefit of making the memory checking fully deterministic. However, the level of control is region based rather than page based, meaning the control is considerably less fine-grained than in the VMSA.

A second simplification is that the PMSA does not support virtual to physical address mapping other than flat address mapping. The physical memory address accessed is the same as the virtual address generated by the processor.

B4.1.1 Protection regions

In a PMSA implementation, you can use CP15 registers to define protection regions in the physical memory map. When describing a PMSA implementation, protection regions are often referred to as regions.

This means the PMSA has the following features:

- For each defined region, CP15 registers specify:
  - — the region size
  - — the base address
  - — the memory attributes, for example, memory type and access permissions.

  Regions of 256 bytes or larger can be split into 8 sub-regions for improved granularity of memory access control.

  The minimum region size supported is IMPLEMENTATION DEFINED.
- Memory region control, requiring read and write access to the region configuration registers, is possible only from privileged modes.
- Regions can overlap. If an address is defined in multiple regions, a fixed priority scheme is used to define the properties of the address being accessed. This scheme gives priority to the region with the highest region number.
- The PMSA can be configured so that an access to an address that is not defined in any region either:
  - — causes a memory abort
  - — if it is a privileged access, uses the default memory map.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B4.1 PMSA について

PMSA は、*メモリ保護ユニット* (MPU) をベースにしています。PMSA によって、第 B3 章*「仮想メモリシステム アーキテクチャ (VMSA)」*で説明されている MMU ベースの VMSA よりもずっと単純なメモリ保護方式が提供されます。この単純化は、ハードウェアとソフトウェアの両方に当てはまります。PMSAv7 プロセッサは、MPU タイプレジスタの存在によって識別されます。*c0、MPU タイプレジスタ (MPUIR)* P. B4-36 を参照して下さい。

主な単純化は、MPU では変換テーブルが使用されないことです。代わりに、システム制御コプロセッサ (CP15) レジスタを使用して、*保護領域*が定義されます。この保護領域により、次のものが必要なくなります。

- 変換テーブルウォークを実行するためのハードウェア
- 変換テーブルのセットアップと保守を行うソフトウェア

保護領域の使用には、メモリのチェックが完全に予測可能になるというメリットがあります。ただし、制御のレベルがページベースではなく領域ベースのため、VMSA よりも粒度の細かさは大幅に低下します。

2 つ目の単純化は、PMSA では、フラット アドレスマッピング以外の仮想アドレスから物理アドレスへのマッピングがサポートされないことです。アクセス対象の物理メモリアドレスは、プロセッサによって生成される仮想アドレスと一致します。

B4.1.1 保護領域

PMSA 実装では、CP15 レジスタを使用して、物理メモリマップの保護領域を定義することができます。PMSA 実装の説明では、多くの場合保護領域を単に領域と呼びます。

PMSA には次のような特徴があります。

- 定義された領域ごとに、CP15 レジスタによって次の情報が指定されます。
  - – 領域サイズ
  - – ベースアドレス
  - – メモリタイプやアクセス許可などのメモリ属性

  256 バイト以上の領域は、8 つのサブ領域に分割することによって、メモリアクセス制御の粒度を高めることができます。

  サポートされる最小の領域サイズは、実装定義です。
- 領域構成レジスタへの読み出し / 書き込みアクセスが必要なメモリ領域制御は、特権モードからのみ可能です。
- 領域はオーバラップさせることができます。1 つのアドレスが複数の領域で定義されている場合は、アクセス対象のアドレスのプロパティを定義するため、固定の優先方式が使用されます。この方式では、領域番号の最も大きい領域が優先されます。
- PMSA は、どの領域にも定義されていないアドレスにアクセスされた場合、次のいずれかが発生するように構成することができます。
  - – メモリアボート
  - – 特権アクセスの場合、デフォルトのメモリマップが使用される。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- All addresses are physical addresses, address translation is not supported.
- Instruction and data address spaces can be either:
  - — unified, so a single region descriptor applies to both instruction and data accesses
  - — separated between different instruction region descriptors and data region descriptors.

When the processor generates a memory access, the MPU compares the memory address with the programmed memory regions:

- If a matching memory region is not found, then:
  - — the access can be mapped onto a background region, see *Using the default memory map as a background region* on page B4-5
  - — otherwise, a Background Fault memory abort is signaled to the processor.
- If a matching memory region is found:
  - — The access permission bits are used to determine whether the access is permitted. If the access is not permitted, the MPU signals a Permissions Fault memory abort. Otherwise, the access proceeds. See *Memory access control* on page B4-9 for a description of the access permission bits.
  - — The memory region attributes are used to determine the memory type, as described in *Memory region attributes* on page B4-11.

B4.1.2 Subregions

A region of the PMSA memory map can be split into eight equal sized, non-overlapping subregions:

- any region size between 256bytes and 4Gbytes supports 8 sub-regions
- region sizes below 256 bytes do not support sub-regions

In the Region Size Register for each region, there is a Subregion disable bit for each subregion. This means that each subregion is either:

- part of the region, if its Subregion disable bit is 0
- not part of the region, if its Subregion disable bit is 1.

If the region size is smaller than 256 bytes then all eight of the Subregion bits are UNK/SBZ.

If a subregion is part of the region then the protection and memory type attributes of the region apply to the subregion. If a subregion is not part of the region then the addresses covered by the subregion do not match as part of the region.

Subregions are not available in versions of the PMSA before PMSAv7.

B4.1.3 Overlapping regions

The MPU can be programmed with two or more overlapping regions. When memory regions overlap, a fixed priority scheme determines the region whose attributes are applied to the memory access. The higher the region number the higher the priority. Therefore, for example, in an implementation that supports eight memory regions, the attributes for region 7 have highest priority and those for region 0 have lowest priority.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- すべてのアドレスは物理アドレスで、アドレス変換はサポートされません。
- 命令とデータのアドレス空間は、次のいずれかです。
  - – 単一の領域記述子が命令アクセスとデータアクセスの両方に適用されるように統一する。
  - – 命令領域記述子とデータ領域記述子とを分離する。

プロセッサでメモリアクセスが生成されると、MPU によって、そのメモリアドレスとプログラムされたメモリ領域とが比較されます。

- 一致するメモリ領域が見つからない場合、次の動作が行われます。
  - – アクセスをバックグラウンド領域にマップすることができます（*デフォルトのメモリマップをバックグラウンド領域として使用する* P. B4-5 を参照）。
  - – それ以外の場合は、バックグラウンドフォールト メモリアボートがプロセッサに通知されます。
- 一致するメモリ領域が見つかった場合、次の動作が行われます。
  - – アクセス許可ビットを使用して、アクセスが許可されているかどうかが判断されます。アクセスが許可されていない場合は、MPU によってアクセス許可フォールト メモリアボートが通知されます。それ以外の場合は、アクセスが続行されます。アクセス許可ビットの説明については、*メモリアクセスの制御* P. B4-9 を参照して下さい。
  - – *メモリ領域の属性* P. B4-11 で説明されているように、メモリ領域属性が、メモリタイプを判断するために使用されます。

B4.1.2 サブ領域

PMSA メモリマップの領域は、サイズが等しく、オーバラップしていない 8 つの同サイズのサブ領域に分割することができます。

- 256 バイト～ 4GB の任意の領域サイズを、8 つのサブ領域に割り当てることができます。
- 256 バイト未満の領域サイズは、サブ領域に割り当てることができません。

各領域の領域サイズレジスタには、サブ領域単位のサブ領域ディセーブルビットが存在します。これは、各サブ領域が次のいずれかであることを意味します。

- 領域の一部（サブ領域ディセーブルビットが 0 の場合）
- 領域の一部ではない（サブ領域ディセーブルビットが 1 の場合）

領域サイズが 256 バイト未満の場合は、8 つのサブ領域ビットのすべてが UNK/SBZ です。

サブ領域が領域の一部の場合は、その領域の保護タイプ属性とメモリタイプ属性が、サブ領域に適用されます。サブ領域が領域の一部でない場合は、サブ領域によってカバーされるアドレスは、領域の一部ではありません。

PMSAv7 以前のバージョンの PMSA では、サブ領域は使用できません。

B4.1.3 オーバラップ領域

MPU は、オーバラップしている複数の領域を使用してプログラムすることができます。メモリ領域がオーバラップしている場合は、固定の優先度付け方式によって、どの領域の属性をメモリアクセスに適用するかが決定されます。領域番号が大きいほど、優先度が高くなります。したがって、例えば、8 つのメモリ領域をサポートしている実装では、領域 7 の属性が最も優先度が高く、領域 0 の属性が最も優先度が低くなります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure B4-1 shows a case where the MPU is programmed with overlapping memory regions.

Figure B4-1 Overlapping memory regions in the MPU

In this example:

- Data region 2 is programmed to be 4KB in size, starting from address `0x3000` with AP[2:0} == 0b010, giving privileged mode full access, User mode read-only access.
- Data region 1 is programmed to be 16KB in size, starting from address `0x0` with AP[2:0} == 0b001, giving privileged mode access only.

If the processor performs a data load from address `0x3010` while in User mode, the address is in both region 1 and region 2. Region 2 has the higher priority, therefore the region 2 attributes apply to the access. This means the load does not abort.

B4.1.4 The background region

Background region refers to a region that matches the entire 4GB physical address map, and has a lower priority than any other region. Therefore, a background region provides the memory attributes for any memory access that does not match any of the defined memory regions.

When the SCTLR.BR bit is set to 0, the MPU behaves as if there is a background region that generates a Background Fault memory abort on any access. This means that any memory access that does not match any of the programmed memory regions generates a Background Fault memory abort. This is the same as the behavior in PMSAv6.

If you want a background region with a different set of memory attributes, you can program region 0 as a 4GB region with the attributes you require. Because region 0 has the lowest priority this region then acts as a background region.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

オーバラップしているメモリ領域を使用してMPUがプログラムされている場合を、図 B4-1 に示します。

図 B4-1 MPU のオーバラップしているメモリ領域

この例では、次の構成を想定します。

- データ領域 2 は、特権モードの完全アクセスとユーザモードの読み出し専用アクセスを示す AP[2:0} == 0b010 を使用して、アドレス `0x3000` から始まる 4KB のサイズにプログラムされています。
- データ領域 1 は、特権モードのアクセスのみを示す AP[2:0} == 0b001 を使用して、アドレス `0x0` から始まる 16KB のサイズにプログラムされています。

プロセッサがユーザモードでアドレス `0x3010` からデータロードを実行すると、そのアドレスは領域 1 と領域 2 の両方に存在します。領域 2 の方が優先度が高いため、領域 2 の属性がアクセスに適用されます。これは、ロードがアボートされないことを意味します。

B4.1.4 バックグラウンド領域

*バックグラウンド領域*は、4GB の物理アドレスマップ全体と一致し、優先度が他のどの領域よりも低い領域を意味します。したがって、バックグラウンド領域では、定義されたどのメモリ領域とも一致しないメモリアクセス用のメモリ属性が提供されます。

SCTLR.BR ビットが 0 にセットされている場合は、MPU が、任意のアクセスでバックグラウンドフォールト メモリアボートが生成されるバックグラウンド領域が存在するかのように動作します。これは、プログラムされたどのメモリ領域とも一致しないメモリアクセスは、すべてバックグラウンドフォールト メモリアボートを生成することを意味します。これは、PMSAv6 での動作と同じです。

別のメモリ属性のセットを持つバックグランド領域が必要な場合は、必要な属性を持つ 4GB の領域として領域 0 をプログラムすることができます。領域 0 の優先度は最低のため、この領域がバックグラウンド領域として機能します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Using the default memory map as a background region

The default memory map is defined in *The default memory map* on page B4-6. Before PMSAv7, the default memory map is used only to define the behavior of memory accesses when the MPU is disabled or not implemented. From PMSAv7, when the SCTLR.BR bit is set to 1, and the MPU is present and enabled:

- the default memory map defines the background region for privileged memory accesses, meaning that a privileged access that does not match any of the programmed memory regions takes the properties defined for that address in the default memory map
- an unprivileged memory access that does not match any of the defined memory regions generates a Background Fault memory abort.

Using the default memory map as the background region means that all of the programmable memory region definitions can be used to define protection regions in the 4GB memory address space.

B4.1.5 Enabling and disabling the MPU

The SCTLR.M bit is used to enable and disable the MPU, see *c1, System Control Register (SCTLR)* on page B4-45. On reset, this bit is cleared to 0, meaning the MPU is disabled after a reset.

Software must program all relevant CP15 registers before enabling the MPU. This includes at least one of:

- setting up at least one memory region
- setting the SCTLR.BR bit to 1, to use the default memory map as a background region, see *Using the default memory map as a background region*.

Synchronization of changes to the CP15 registers is discussed in *Changes to CP15 registers and the memory order model* on page B4-28. These considerations apply to any change that enables or disables the MPU or the caches.

Behavior when the MPU is disabled

When the MPU is disabled:

- Instruction accesses use the default memory map and attributes shown in Table B4-1 on page B4-6. An access to a memory region with the Execute Never attribute generates a Permission fault, see *The Execute Never (XN) attribute and instruction prefetching* on page B4-10. No other permission checks are performed. Additional control of the cacheability is made by:
  - — the SCTLR.I bit if separate instruction and data caches are implemented
  - — the SCTLR.C bit if unified caches are implemented.
- Data accesses use the default memory map and attributes shown in Table B4-2 on page B4-7. No memory access permission checks are performed, and no aborts can be generated.
- Program flow prediction functions as normal, controlled by the value of the SCTLR.Z bit, see *c1, System Control Register (SCTLR)* on page B4-45.
- All of the CP15 cache operations work as normal.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

デフォルトのメモリマップをバックグラウンド領域として使用する

デフォルトのメモリマップについては、*デフォルトのメモリマップ* P. B4-6 で説明します。PMSAv7 以前は、デフォルトのメモリマップは、MPU が禁止されている、または実装されていない場合のメモリアクセスの動作を定義するためにのみ使用されます。PMSAv7 およびそれ以降は、SCTLR.BR ビットが 1 にセットされ、MPU が存在し許可されている場合に、次のように使用されます。

- デフォルトのメモリマップによって、特権メモリアクセスのバックグラウンド領域が定義されます。つまり、プログラムされているどのメモリ領域とも一致しない特権アクセスは、デフォルトのメモリマップでそのアドレスについて定義されたプロパティを使用します。
- 定義されているどのメモリ領域とも一致しない非特権メモリアクセスは、バックグラウンドフォールト メモリアボートを生成します。

バックグラウンド領域としてのデフォルトのメモリマップを使用すると、プログラム可能なすべてのメモリ領域定義を使用して、4GB のメモリ領域空間内に保護領域を定義できます。

B4.1.5 MPU の許可と禁止

SCTLR.M ビットは、MPU を許可または禁止するために使用されます。*c1、システム制御レジスタ (SCTLR)* P. B4-45 を参照して下さい。このビットは、リセット時に 0 にクリアされます。これにより、MPU はリセット後に禁止されます。

ソフトウェアで MPU を許可する前に、関連するすべての CP15 レジスタをプログラムする必要があります。これには、少なくとも次のいずれかが含まれます。

- 少なくとも 1 つのメモリ領域のセットアップ
- デフォルトのメモリマップをバックグラウンド領域として使用するため、SCTLR.BR ビットを 1 にセットする（*デフォルトのメモリマップをバックグラウンド領域として使用する*を参照）。

CP15 レジスタの変更の同期については、*CP15 レジスタの変更とメモリオーダ モデル* P. B4-28 で説明します。これらの考慮事項は、MPU またはキャッシュを稼動または非稼動にするすべての変更に当てはまります。

MPU が稼動していない場合の動作

MPU が稼動していない場合、次の規則が適用されます。

- 命令アクセスでは、P. B4-6 の表 B4-1 に示すデフォルトのメモリマップと属性が使用されます。実行不可属性を持つメモリ領域へのアクセスは、アクセス許可フォールトを生成します（*実行不可 (XN) 属性と命令のプリフェッチ* P. B4-10 を参照）。その他のアクセス許可チェックは実行されません。次のビットは、キャッシュ可否について追加の制御を行います。
  - – 独立した命令キャッシュとデータキャッシュが実装されている場合の、SCTLR.I ビット
  - – 統一キャッシュが実装されている場合の、SCTLR.C ビット
- データアクセスでは、P. B4-7 の表 B4-2 に示すデフォルトのメモリマップと属性が使用されます。メモリアクセス許可チェックは実行されず、アボートは生成されません。
- SCTLR.Z ビットの値によって制御されるプログラムフロー予測機能は、通常に機能します（*c1、システム制御レジスタ (SCTLR)* P. B4-45 を参照）。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- Instruction and data prefetch operations work as normal, based on the default memory map:
  - — Data prefetch operations have no effect if the data cache is disabled
  - — Instruction prefetch operations have no effect if the instruction cache is disabled.
- The Outer memory attributes are the same as those for the Inner memory system.

The default memory map

The PMSAv7 default memory map is fixed and not configurable, and is shown in:

- Table B4-1 for the instruction access attributes
- Table B4-2 on page B4-7 for the data access attributes.

The regions of the default memory map are identical in both tables. The information about the memory map is split into two tables only to improve the presentation of the information.

Table B4-1 Default memory map, showing instruction access attributes

Address range	HIVECS	Instruction memory type		Execute Never, XN
		Caching enabled[a]	Caching disabled[a]	
0xFFFFFFFF -0xF0000000	0	Not applicable	Not applicable	Execute Never
0xFFFFFFFF - 0xF0000000	1[b]	Normal, Non-cacheable	Normal, Non-cacheable	Execution permitted
0xEFFFFFFF - 0xC0000000	X	Not applicable	Not applicable	Execute Never
0xBFFFFFFF - 0xA0000000	X	Not applicable	Not applicable	Execute Never
0x9FFFFFFF - 0x80000000	X	Not applicable	Not applicable	Execute Never
0x7FFFFFFF - 0x60000000	X	Normal, Non-shareable, Write-Through Cacheable	Normal, Non-shareable, Non-cacheable	Execution permitted
0x5FFFFFFF - 0x40000000	X	Normal, Non-shareable, Write-Through Cacheable	Normal, Non-shareable, Non-cacheable	Execution permitted
0x3FFFFFFF - 0x00000000	X	Normal, Non-shareable, Write-Through Cacheable	Normal, Non-shareable, Non-cacheable	Execution permitted

a. When separate instruction and data caches are implemented, caching is enabled for instruction accesses if the instruction caches are enabled. When unified caches are implemented caching is enabled if the data or unified caches are enabled. See the descriptions of the C and I bits in *c1, System Control Register (SCTLR)* on page B4-45.

b. Use of HIVECS == 1 is deprecated in PMSAv7, see *Exception vectors and the exception base address* on page B1-30.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- すべての CP15 キャッシュ操作は通常に動作します。
- 命令およびデータのプリフェッチ操作は、デフォルトのメモリマップに基づいて通常に動作します。
  - ー データキャッシュが稼動していない場合、データプリフェッチ操作は効力を持ちません。
  - ー 命令キャッシュが稼動していない場合、命令プリフェッチ操作は効力を持ちません。
- 外部メモリ属性は、内部メモリシステムの属性と同じです。

デフォルトのメモリマップ

PMSAv7 のデフォルトのメモリマップは固定で、構成はできません。このメモリマップを次の表に示します。

- 表 B4-1（命令アクセス属性の場合）
- P. B4-7 の表 B4-2 （データアクセス属性の場合）

どちらに表でも、デフォルトのメモリマップの領域は全く同じです。メモリマップに関する情報は、見やすくする目的でのみ 2 つの表に分割されています。

表 B4-1 メモリアクセス属性を示すデフォルトのメモリマップ

アドレスの範囲	HIVECS	命令メモリタイプ		実行不可、XN
		キャッシュ許可[a]	キャッシュ禁止[a]	
0xFFFFFFFF 〜 0xF0000000	0	該当せず	該当せず	実行不可
0xFFFFFFFF 〜 0xF0000000	1[b]	ノーマル、キャッシュ不可	ノーマル、キャッシュ不可	実行可能
0xEFFFFFFF 〜 0xC0000000	X	該当せず	該当せず	実行不可
0xBFFFFFFF 〜 0xA0000000	X	該当せず	該当せず	実行不可
0x9FFFFFFF 〜 0x80000000	X	該当せず	該当せず	実行不可
0x7FFFFFFF 〜 0x60000000	X	ノーマル、共有不可、ライトスルーキャッシュ可能	ノーマル、共有不可、キャッシュ不可	実行可能
0x5FFFFFFF 〜 0x40000000	X	ノーマル、共有不可、ライトスルーキャッシュ可能	ノーマル、共有不可、キャッシュ不可	実行可能
0x3FFFFFFF 〜 0x00000000	X	ノーマル、共有不可、ライトスルーキャッシュ可能	ノーマル、共有不可、キャッシュ不可	実行可能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B4-2 Default memory map, showing data access attributes

Address range	Data memory type	
	Caching enabled[a]	Caching disabled
0xFFFFFFFF - 0xC0000000	Strongly-ordered	Strongly-ordered
0xBFFFFFFF - 0xA0000000	Shareable Device	Shareable Device
0x9FFFFFFF - 0x80000000	Non-shareable Device	Non-shareable Device
0x7FFFFFFF - 0x60000000	Normal, Shareable, Non-cacheable	Normal, Shareable, Non-cacheable
0x5FFFFFFF - 0x40000000	Normal, Non-shareable, Write-Through Cacheable	Normal, Shareable, Non-cacheable
0x3FFFFFFF - 0x00000000	Normal, Non-shareable, Write-Back, Write-Allocate Cacheable	Normal, Shareable, Non-cacheable

a. Caching is enabled for data accesses if the data or unified caches are enabled. See the description of the C bit in *c1, System Control Register (SCTLR)* on page B4-45.

Behavior of an implementation that does not include an MPU

If a PMSAv7 implementation does not include an MPU, it must adopt the default memory map behavior described in *Behavior when the MPU is disabled* on page B4-5.

A PMSAv7 implementation that does not include an MPU is identified by an MPU Type Register entry that shows a Unified MPU with zero Data or Unified regions, see *c0, MPU Type Register (MPUIR)* on page B4-36.

B4.1.6 Finding the minimum supported region size

You can use the DRBAR to find the minimum region size supported by an implementation, by following this procedure:

1. Write a valid memory region number to the RGNR. Normally you use region number 0, because this is always a valid region number.
2. Write the value 0xFFFFFFFC to the DRBAR. This value sets all valid bits in the register to 1.
3. Read back the value of the DRBAR. In the returned value the least significant bit set indicates the resolution of the selected region. If the least significant bit set is bit M the resolution of the region is 2^M bytes.

If the MPU implements separate data and instruction regions this process gives the minimum size for data regions. To find the minimum size for instruction regions, use the same procedure with the IRBAR.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

a. 独立した命令キャッシュとデータキャッシュが実装されている場合は、命令キャッシュが稼動していれば、命令アクセスに対してキャッシュが可能です。統一キャッシュが実装されている場合は、データまたは統一キャッシュが可能であれば、キャッシュが可能です。*c1、システム制御レジスタ (SCTLR)* P. B4-45 の C ビットと I ビットに関する説明を参照して下さい。
b. PMSAv7 では HIVECS == 1 の使用が非推奨です（*例外ベクタと例外ベースアドレス* P. B1-30 を参照）。

表 B4-2 データアクセス属性を示すデフォルトのメモリマップ

アドレスの範囲	データメモリ タイプ	
	キャッシュ許可[a]	キャッシュ禁止
0xFFFFFFFF 〜 0xC0000000	ストロングリオーダ	ストロングリオーダ
0xBFFFFFFF 〜 0xA0000000	共有可能デバイス	共有可能デバイス
0x9FFFFFFF 〜 0x80000000	共有不可デバイス	共有不可デバイス
0x7FFFFFFF 〜 0x60000000	ノーマル、共有可能、キャッシュ不可	ノーマル、共有可能、キャッシュ不可
0x5FFFFFFF 〜 0x40000000	ノーマル、共有不可、ライトスルーキャッシュ可能	ノーマル、共有可能、キャッシュ不可
0x3FFFFFFF 〜 0x00000000	ノーマル、共有不可、ライトバック、書き込み割り当てキャッシュ可能	ノーマル、共有可能、キャッシュ不可

a. データまたは統一キャッシュが可能な場合は、データアクセスに対するキャッシュが可能です。*c1、システム制御レジスタ (SCTLR)* P. B4-45 の C ビットに関する説明を参照して下さい。

MPU を含まない実装の動作

PMSAv7 実装に MPU が含まれていない場合は、*MPU が稼動していない場合の動作* P. B4-5 で説明されているデフォルトのメモリマップの動作を採用する必要があります。

MPU を含まない PMSAv7 実装は、データ領域または統一領域が 0 の統一 MPU を示す、MPU タイプレジスタのエントリによって識別されます。*c0、MPU タイプレジスタ (MPUIR)* P. B4-36 を参照して下さい。

B4.1.6 サポートされている最小領域サイズの判定

DRBAR を使用すると、次の手順に従って、実装でサポートされている最小領域サイズを判定できます。

1. 有効なメモリ領域番号を RGNR に書き込みます。通常は、常に有効な領域番号である領域番号 0 を使用します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For more information about the registers used see:

- *c6, MPU Region Number Register (RGNR)* on page B4-66
- *c6, Data Region Base Address Register (DRBAR)* on page B4-60
- *c6, Instruction Region Base Address Register (IRBAR)* on page B4-61.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

2. 値 0xFFFFFFFC を DRBAR に書き込みます。この値によって、レジスタ内のすべての有効なビットが 1 にセットされます。

3. DRBAR の値を読み出します。戻り値でセットされている最下位ビットが、選択された領域の解像度を表します。セットされている最下位ビットがビット M の場合、領域の解像度は 2^M バイトです。

MPU に独立したデータ領域と命令領域が実装されている場合は、このプロセスによってデータ領域の最小サイズが提供されます。命令領域の最小サイズを判定するには、IRBAR を使用して同じ手順を実行します。

使用されるレジスタの詳細については、次のセクションを参照して下さい。

- *c6、MPU 領域番号レジスタ (RGNR)* P. B4-66
- *c6、データ領域ベースアドレス レジスタ (DRBAR)* P. B4-60
- *c6、命令領域ベースアドレス レジスタ (IRBAR)* P. B4-61

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.2 Memory access control

Access to a memory region is controlled by the access permission bits for each region, held in the DRACR and IRACR. For descriptions of the registers see:

- *c6, Data Region Access Control Register (DRACR)* on page B4-64
- *c6, Instruction Region Access Control Register (IRACR)* on page B4-65.

B4.2.1 Access permissions

Access permission bits control access to the corresponding memory region. If an access is made to an area of memory without the required permissions, a Permission fault is generated. In the appropriate Region Access Control Register:

- the AP bits determine the access permissions
- the XN bit provides an additional permission bit for instruction fetches.

The access permissions are a three-bit field, DRACR.AP[2:0] or IRACR.AP[2:0]. Table B4-3 shows the possible values of this field.

Table B4-3 Access permissions

AP[2:0]	Privileged permissions	User permissions	Description
000	No access	No access	All accesses generate a Permission fault
001	Read/Write	No access	All User mode accesses generate Permission faults
010	Read/Write	Read-only	User mode write accesses generate Permission faults
011	Read/Write	Read/Write	Full access
100	UNPREDICTABLE	UNPREDICTABLE	Reserved
101	Read-only	No Access	Privileged read-only, all other accesses generate Permission faults
110	Read-only	Read-only	All write accesses generate Permission faults.
111	UNPREDICTABLE	UNPREDICTABLE	Reserved

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.2 メモリアクセスの制御

メモリ領域へのアクセスは、DRACR および IRACR に保持されている、領域ごとのアクセス許可ビットによって制御されます。これらのレジスタの説明については、次のセクションを参照して下さい。

- *c6、データ領域アクセス制御レジスタ (DRACR)* P. B4-64
- *c6、命令領域アクセス制御レジスタ (IRACR)* P. B4-65

B4.2.1 アクセス許可

アクセス許可ビットは、対応するメモリ領域へのアクセスを制御します。必要な許可なしにメモリ領域へアクセスを行うと、アクセス許可フォールトが生成されます。該当する領域アクセス制御レジスタでは、次の処理が行われます。

- AP ビットによってアクセス許可が決定されます。
- XN ビットによって、命令フェッチに関する他の許可ビットが提供されます。

アクセス許可は、3 ビットのフィールド、DRACR.AP[2:0] または IRACR.AP[2:0] です。このフィールドの可能な値を、表 B4-3 に示します。

表 B4-3 アクセス許可

AP[2:0]	特権アクセス許可	ユーザアクセス許可	説明
000	アクセス不可	アクセス不可	すべてのアクセスでアクセス許可フォールトが発生
001	読み出し / 書き込み	アクセス不可	すべてのユーザモード アクセスでアクセス許可フォールトが発生
010	読み出し / 書き込み	読み出し専用	ユーザモード書き込みアクセスでアクセス許可フォールトが発生
011	読み出し / 書き込み	読み出し / 書き込み	完全アクセス
100	予測不能	予測不能	予約
101	読み出し専用	アクセス不可	特権読み出し専用、他のすべてのアクセスでアクセス許可フォールトが発生
110	読み出し専用	読み出し専用	すべての書き込みアクセスでアクセス許可フォールトが発生
111	予測不能	予測不能	予約

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The Execute Never (XN) attribute and instruction prefetching

Each memory region can be tagged as not containing executable code. If the *Execute never* (XN) bit is set to 1, any attempt to execute an instruction in that region results in a Permission fault, and the implementation must not access the region to prefetch instructions speculatively. If the XN bit is 0, code can execute from that memory region.

Note

The XN bit acts as an additional permission check. The address must also have a valid read access permission.

In ARMv7, all regions of memory that contain read-sensitive peripherals must be marked as XN to avoid the possibility of a speculative prefetch accessing the locations.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

実行不可 (XN) 属性と命令のプリフェッチ

メモリ領域ごとに、実行可能コードが含まれていないことを示すタグを付けることができます。*実行不可*(XN) ビットが 1 に設定されている場合は、その領域内で命令の実行を試みるとアクセス許可フォールトが発生するため、実装で投機的にその領域にアクセスして命令をプリフェッチしないようにする必要があります。XN ビットが 0 の場合は、そのメモリ領域からコードを実行することができます。

注

XN ビットは、補助的なアクセス許可チェックとして機能します。アドレスには有効な読み出しアクセス許可も必要です。

ARMv7 では、読み出しで影響を受けるペリフェラルを含むすべてのメモリ領域は XN としてマークし、その場所への投機的なプリフェッチアクセスの可能性を避ける必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.3 Memory region attributes

Each memory region has an associated set of memory region attributes. These control accesses to the caches, how the write buffer is used, and whether the memory region is Shareable and therefore is guaranteed by hardware to be coherent. These attributes are encoded in the C, B, TEX[2:0] and S bits of the appropriate Region Access Control Register.

Note

The *Bufferable* (B), *Cacheable* (C), and *Type Extension* (TEX) bit names are inherited from earlier versions of the architecture. These names no longer adequately describe the function of the B, C, and TEX bits.

B4.3.1 C, B, and TEX[2:0] encodings

The TEX[2:0] field must be considered with the C and B bits to give a five bit encoding of the access attributes for an MPU memory region. Table B4-4 shows these encodings.

For Normal memory regions, the S (Shareable) bit gives more information about whether the region is Shareable. A Shareable region can be shared by multiple processors. A Normal memory region is Shareable if the S bit for the region is set to 1. For other memory types, the value of the S bit is ignored.

Table B4-4 C, B and TEX[2:0] encodings

TEX[2:0]	C	B	Description	Memory type	Shareable?
000	0	0	Strongly-ordered.	Strongly-ordered	Shareable
000	0	1	Shareable Device.	Device	Shareable
000	1	0	Outer and Inner Write-Through, no Write-Allocate.	Normal	S bit[a]
000	1	1	Outer and Inner Write-Back, no Write-Allocate.	Normal	S bit[a]
001	0	0	Outer and Inner Non-cacheable.	Normal	S bit[a]
001	0	1	Reserved.	-	-
001	1	0	IMPLEMENTATION DEFINED.	IMP. DEF.[b]	IMP. DEF.[b]
001	1	1	Outer and Inner Write-Back, Write-Allocate.	Normal	S bit[a]
010	0	0	Non-shareable Device.	Device	Non-shareable
010	0	1	Reserved.	-	-
010	1	X	Reserved.	-	-

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.3 メモリ領域の属性

メモリ領域ごとに、一連のメモリ領域属性が関連付けられています。これらの属性によって、キャッシュに対するアクセス、ライトバッファの使用方法、およびメモリ領域が共有可能かどうかが制御されるため、ハードウェアによって整合性が保証されます。これらの属性は、該当する領域アクセス制御レジスタの C、B、TEX[2:0]、S ビットにエンコードされます。

注

バッファ可能(B)、*キャッシュ可能*(C)、*タイプ拡張*(TEX) の各ビット名は、以前のバージョンのアーキテクチャから継承されたものです。これらの名前は、現在では B、C、TEX ビットの機能を適切に表現しなくなっています。

B4.3.1 C、B、TEX[2:0] のエンコード

TEX[2:0] フィールドは、C ビット、B ビットと組み合わせて、MPU メモリ領域のアクセス属性の 5 ビットエンコードを提供する必要があります。これらのエンコードを表 B4-4 に示します。

ノーマルメモリ領域の場合は、S（共有可能）ビットによって、領域が共有可能かどうかに関する追加の情報が提供されます。共有可能な領域は、複数のプロセッサで共有できます。ノーマルメモリ領域は、その S ビットが 1 にセットされている場合に共有可能です。他のメモリタイプでは、S ビットの値が無視されます。

表 B4-4 C、B、TEX[2:0] のエンコード

TEX[2:0]	C	B	説明	メモリタイプ	共有可能か
000	0	0	ストロングリオーダ	ストロングリオーダ	共有可能
000	0	1	共有可能デバイス	デバイス	共有可能
000	1	0	外部および内部ライトスルー、書き込み割り当てなし	ノーマル	S ビット[a]
000	1	1	外部および内部ライトバック、書き込み割り当てなし	ノーマル	S ビット[a]
001	0	0	外部および内部キャッシュ不可	ノーマル	S ビット[a]
001	0	1	予約	–	–
001	1	0	実装定義	IMP. DEF.[b]	IMP. DEF.[b]
001	1	1	外部および内部ライトバック、書き込み割り当て	ノーマル	S ビット[a]
010	0	0	共有不可デバイス	デバイス	共有不可
010	0	1	予約	–	–
010	1	X	予約	–	–

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B4-4 C, B and TEX[2:0] encodings (continued)

TEX[2:0]	C	B	Description		Memory type	Shareable?
011	X	X	Reserved.		-	-
1BB	A	A	Cacheable memory:	AA = Inner attribute[c] BB = Outer policy	Normal	S bit[a]

a. Region is Shareable if S == 1, and Non-shareable if S == 0.
b. IMP. DEF. = IMPLEMENTATION DEFINED.
c. For more information see *Cacheable memory attributes*.

For an explanation of Normal, Strongly-ordered and Device memory types, and the Shareable attribute, see *Memory types and attributes and the memory order model* on page A3-24.

Cacheable memory attributes

When TEX[2] == 1, the memory region is Cacheable memory, and the rest of the encoding defines the Inner and Outer cache attributes:

TEX[1:0] defines the Outer cache attribute
C,B defines the Inner cache attribute

The same encoding is used for the Outer and Inner cache attributes. Table B4-5 shows the encoding.

Table B4-5 Inner and Outer cache attribute encoding

Memory attribute encoding	Cache attribute
00	Non-cacheable
01	Write-Back, Write-Allocate
10	Write-Through, no Write-Allocate
11	Write-Back, no Write-Allocate

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 B4-4 C、B、TEX[2:0] のエンコード（続き）

TEX[2:0]	C	B	説明		メモリタイプ	共有可能か
011	X	X	予約		–	–
1BB	A	A	キャッシュ可能メモリ	AA = 内部属性[c] BB = 外部ポリシー	ノーマル	S ビット[a]

a. 領域は、S == 1 の場合に共有可能で、S == 0 の場合に共有不可です。
b. IMP. DEF. = 実装定義
c. 詳細については、*キャッシュ可能メモリ属性*を参照して下さい。

ノーマル、ストロングオーダ、デバイスの各メモリタイプと共有可能属性の説明については、*メモリのタイプおよび属性とメモリオーダ モデル* P. A3-24 を参照して下さい。

キャッシュ可能メモリ属性

TEX[2] == 1 の場合、メモリ領域はキャッシュ可能メモリで、残りのエンコードで内部と外部のキャッシュ属性が定義されます。

TEX[1:0] 外部キャッシュ属性を定義します。
C、B 内部キャッシュ属性を定義します。

内部と外部のキャッシュ属性では同じエンコードが使用されます。エンコードを表 B4-5 に示します。

表 B4-5 内部と外部のキャッシュ属性のエンコード

メモリ属性のエンコード	キャッシュ属性
00	キャッシュ不可
01	ライトバック、書き込み割り当て
10	ライトスルー、書き込み割り当てなし
11	ライトバック、書き込み割り当てなし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.4 PMSA memory aborts

The mechanisms that cause the ARM processor to take an exception because of a memory access are:

MPU fault The MPU detects an access restriction and signals the processor.

External abort A memory system component other than the MPU signals an illegal or faulting external memory access.

The exception taken is a Prefetch Abort exception if either of these occurs synchronously on an instruction fetch, and a Data Abort exception otherwise.

Collectively these mechanisms are called *aborts*. The different abort mechanisms are described in:

- *MPU faults*
- *External aborts* on page B4-15.

An access that causes an abort is said to be *aborted*, and uses the *Fault Address Registers* (FARs) and *Fault Status Registers* (FSRs) to record context information. The FARs and FSRs are described in *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18.

Also, a debug exception can cause the processor to take a Prefetch Abort exception or a Data Abort exception, and to update the FARs and FSRs. For details see Chapter C4 *Debug Exceptions* and *Debug event prioritization* on page C3-43.

B4.4.1 MPU faults

The MPU checks the memory accesses required for instruction fetches and for explicit memory accesses:

- if an instruction fetch faults it generates a Prefetch Abort exception
- if an explicit memory access faults it generates a Data Abort exception.

For more information about Prefetch Abort exceptions and Data Abort exceptions see *Exceptions* on page B1-30.

MPU faults are always synchronous. For more information, see *Terminology for describing exceptions* on page B1-4.

When the MPU generates an abort for a region of memory, no memory access is made if that region is or could be marked as Strongly-ordered or Device.

The MPU can generate three types of fault, described in the subsections:

- *Alignment fault* on page B4-14
- *Background fault* on page B4-14
- *Permission fault* on page B4-14.

The MPU fault checking sequence on page B4-15 describes the fault checking sequence.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.4 PMSA メモリアボート

メモリアクセスが原因で ARM プロセッサで例外が取得される機構は次のとおりです。

MPU フォールト MPU がアクセス制限を検出して、プロセッサに通知します。

外部アボート MPU 以外のメモリシステム コンポーネントが、不正または不良な外部メモリアクセスを通知します。

取得される例外は、このどちらかが命令フェッチと同期して発生した場合はプリフェッチアボート例外で、それ以外の場合はデータアボート例外です。

これらの機構を総称して*アボート*と呼びます。次のセクションで、さまざまなアボート機構について説明します。

- *MPU フォールト*
- *外部アボート* P. B4-15

アボートを引き起こしたアクセスは、*アボートされた*と呼ばれ、*フォールトアドレス レジスタ* (FAR) と*フォールトステータス レジスタ* (FSR) を使用してコンテキスト情報が記録されます。FAR と FSR については、*PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4-18 で説明します。

また、デバッグ例外によっても、プロセッサでプリフェッチアボート例外またはデータアボート例外が取得され、FAR と FSR が更新される可能性があります。詳細については、第 C4 章*「デバッグ例外」*と*デバッグイベントの優先度付け* P. C3-43 を参照して下さい。

B4.4.1 MPU フォールト

MPU は、命令フェッチと明示的なメモリアクセスに必要なメモリアクセスをチェックします。

- 命令フェッチが失敗した場合は、プリフェッチアボート例外が生成されます。
- 明示的なメモリアクセスが失敗した場合は、データアボート例外が生成されます。

プリフェッチアボート例外とデータアボート例外の詳細については、*例外* P. B1-30 を参照して下さい。

MPU フォールトは必ず同期です。詳細については、*例外に関する用語* P. B1-4 を参照して下さい。

MPU でメモリ領域についてアボートが生成されると、その領域がストロングオーダまたはデバイスとしてマークされている場合、またはマークされている可能性がある場合は、メモリアクセスが実行されません。

MPU は 3 種類のフォールトを生成することができます。これらについては、次のサブセクションで説明します。

- *アライメントフォールト* P. B4-14
- *バックグラウンドフォールト* P. B4-14
- *アクセス許可フォールト* P. B4-14

フォールトチェック シーケンスについては、*MPU のフォールトチェック シーケンス* P. B4-15 で説明します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Alignment fault

The ARMv7 memory architecture requires support for strict alignment checking. This checking is controlled by the SCTLR.A bit, see *c1, System Control Register (SCTLR)* on page B4-45. For details of when Alignment faults are generated see *Unaligned data access* on page A3-5.

Background fault

If the memory access address does not match one of the programmed MPU memory regions, and the default memory map is not being used, a Background Fault memory abort is generated.

Background faults cannot occur on any cache or branch predictor maintenance operation.

Permission fault

The access permissions, defined in *Memory access control* on page B4-9, are checked against the processor memory access. If the access is not permitted, a Permission Fault memory abort is generated.

Permission faults cannot occur on cache or branch predictor maintenance operation.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アライメントフォールト

ARMv7 メモリアーキテクチャでは、厳密なアライメントチェックをサポートする必要があります。このチェックは SCTLR.A ビットによって制御されます。*c1、システム制御レジスタ (SCTLR)* P. B4-45 を参照して下さい。アライメントフォールトが生成されるタイミングの詳細については、*アンアラインド データアクセス* P. A3-5 を参照して下さい。

バックグラウンドフォールト

メモリアクセス アドレスがプログラムされた MPU メモリ領域の 1 つと一致せず、デフォルトのメモリマップが使用されていない場合は、バックグラウンドフォールト メモリアボートが生成されます。

バックグラウンドフォールトは、キャッシュまたは分岐予測器の保守操作では発生しません。

アクセス許可フォールト

メモリアクセスの制御 P. B4-9 で定義されているアクセス許可は、プロセッサのメモリアクセスに対してチェックされます。アクセスが許可されていない場合は、アクセス許可フォールト メモリアボートが生成されます。

アクセス許可フォールトは、キャッシュまたは分岐予測器の保守操作では発生しません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The MPU fault checking sequence

Figure B4-2 shows the MPU fault checking sequence, when the MPU is enabled.

Figure B4-2 MPU fault checking sequence

B4.4.2 External aborts

External memory errors are defined as errors that occur in the memory system other than those that are detected by the MPU or Debug hardware. They include parity errors detected by the caches or other parts of the memory system. An external abort is one of:

- synchronous
- precise asynchronous
- imprecise asynchronous.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

MPU のフォールトチェック シーケンス

MPU が稼動している場合の MPU のフォールトチェック シーケンスを、図 B4-2 に示します。

図 B4-2 MPU のフォールトチェック シーケンス

B4.4.2 外部アボート

外部メモリエラーは、メモリシステムで発生するエラーのうち、MPU またはデバッグハードウェアで検出されるエラーを除くものとして定義されます。キャッシュまたはメモリシステムの他の部分で検出されたパリティエラーも含まれます。外部アボートには次の種類があります。

- 同期
- 正確な非同期
- 不正確な非同期

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For more information, see *Terminology for describing exceptions* on page B1-4.

The ARM architecture does not provide a method to distinguish between precise asynchronous and imprecise asynchronous aborts.

The ARM architecture handles asynchronous aborts in a similar way to interrupts, except that they are reported to the processor using the Data Abort exception. Setting the CPSR.A bit to 1 masks asynchronous aborts, see *Program Status Registers (PSRs)* on page B1-14.

Normally, external aborts are rare. An imprecise asynchronous external abort is likely to be fatal to the process that is running. An example of an event that might cause an external abort is an uncorrectable parity or ECC failure on a Level 2 memory structure.

It is IMPLEMENTATION DEFINED which external aborts, if any, are supported.

PMSAv7 permits external aborts on data accesses and instruction fetches to be either synchronous or asynchronous. The DFSR indicates whether the external abort is synchronous or asynchronous, see *c5, Data Fault Status Register (DFSR)* on page B4-55.

Note

Because imprecise external aborts are normally fatal to the process that caused them, ARM recommends that implementations make external aborts precise wherever possible.

More information about possible external aborts is given in the subsections:

- *External abort on instruction fetch*
- *External abort on data read or write*
- *Parity error reporting* on page B4-17.

For information about how external aborts are reported see *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18.

External abort on instruction fetch

An external abort on an instruction fetch can be either synchronous or asynchronous. A synchronous external abort on an instruction fetch is taken precisely.

An implementation can report the external abort asynchronously from the instruction that it applies to. In such an implementation these aborts behave essentially as interrupts. They are masked by the CPSR.A bit when it is set to 1, otherwise they are reported using the Data Abort exception.

External abort on data read or write

Externally generated errors during a data read or write can be either synchronous or asynchronous.

An implementation can report the external abort asynchronously from the instruction that generated the access. In such an implementation these aborts behave essentially as interrupts. They are masked by the CPSR.A bit when it is set to 1, otherwise they are reported using the Data Abort exception.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

詳細については、*例外に関する用語* P. B1-4 を参照して下さい。

ARM アーキテクチャでは、正確な非同期アボートと不正確な非同期アボートとを区別するための方法は提供されません。

ARM アーキテクチャでは、非同期アボートは、データアボート例外を使用してプロセッサに報告されること以外は、割り込みと同様の方法で処理されます。CPSR.A ビットを 1 にセットすると、非同期アボートがマスクされます。*プログラムステータス レジスタ (PSR)* P. B1-14 を参照して下さい。

通常は、外部アボートが発生することはほとんどありません。不正確な非同期外部アボートは、実行中のプロセスにとって致命的な場合があります。外部アボートを引き起こす可能性のあるイベントの例としては、レベル 2 メモリ構造上の修正不可能なパリティや ECC エラーがあります。

外部アボートをサポートするかどうか、どれをサポートするかは実装定義です。

PMSAv7 では、データアクセスおよび命令フェッチでの外部アボートを、同期と非同期のどちらにもすることができます。DFSR によって、外部アボートが同期と非同期のどちらかが示されます。*c5、データフォールト ステータスレジスタ (DFSR)* P. B4-55 を参照して下さい。

注

不正確な外部アボートはそれを引き起こしたプロセスにとって一般に致命的なため、実装では外部アボートを可能な限り正確にすることをお勧めします。

可能な外部アボートの詳細については、次のサブセクションで説明します。

- *命令フェッチでの外部アボート*
- *データの読み出し / 書き込みでの外部アボート*
- *パリティエラーの報告* P. B4-17

外部アボートがどのように報告されるかについては、*PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4-18 を参照して下さい。

命令フェッチでの外部アボート

命令フェッチでの外部アボートは、同期と非同期のどちらの可能性もあります。命令フェッチでの同期外部アボートは正確に取得されます。

この外部アボートは、実装で、該当する命令とは非同期に報告することができます。このような実装では、これらのアボートが本質的に割り込みとして動作します。この外部アボートは、CPSR.A ビットが 1 にセットされた場合にマスクされ、それ以外の場合はデータアボート例外を使用して報告されます。

データの読み出し / 書き込みでの外部アボート

データの読み出しまたは書き込み時に外部で生成されるエラーは、同期と非同期のどちらの可能性もあります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Parity error reporting

The ARM architecture supports the reporting of both synchronous and asynchronous parity errors from the cache systems. It is IMPLEMENTATION DEFINED what parity errors in the cache systems, if any, result in synchronous or asynchronous parity errors.

A fault status code is defined for reporting parity errors, see *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18. However when parity error reporting is implemented it is IMPLEMENTATION DEFINED whether the assigned fault status code or another appropriate encoding is used to report parity errors.

For all purposes other than the fault status encoding, parity errors are treated as external aborts.

B4.4.3 Prioritization of aborts

For synchronous aborts, *Debug event prioritization* on page C3-43 describes the relationship between debug events, MPU faults and external aborts.

In general, the ARM architecture does not define when asynchronous events are taken, and therefore the prioritization of asynchronous events is IMPLEMENTATION DEFINED.

Note

A special requirement applies to asynchronous watchpoints, see *Debug event prioritization* on page C3-43.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

この外部アボートは、実装で、アクセスを生成した命令とは非同期に報告することができます。このような実装では、これらのアボートが本質的に割り込みとして動作します。この外部アボートは、CPSR.A ビットが 1 にセットされた場合にマスクされ、それ以外の場合はデータアボート例外を使用して報告されます。

パリティエラーの報告

ARM アーキテクチャでは、キャッシュシステムからの同期と非同期両方のパリティエラーの報告をサポートしています。キャッシュシステムのどのパリティエラーが同期または非同期のパリティエラーになるかは、実装定義です。

パリティエラーを報告するためのフォールトステータス コードが定義されています。*PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4-18 を参照して下さい。ただし、パリティエラー報告機能が実装されていても、パリティエラーの報告に、割り当てられたフォールトステータス コードまたは他の適切なエンコードのどちらが使用されるかは実装定義です。

フォールトステータスのエンコード以外のすべての目的には、パリティエラーが外部アボートとして扱われます。

B4.4.3 アボートの優先順位付け

同期アボートに関するデバッグイベント、MPU フォールト、外部アボートの関係については、*デバッグイベントの優先度付け* P. C3-43 で説明します。

通常、ARM アーキテクチャでは、非同期イベントの取得タイミングが定義されていないため、非同期イベントの優先順位付けは実装定義です。

注

非同期ウォッチポイントには特殊な要件が適用されます。*デバッグイベントの優先度付け* P. C3-43 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.5 Fault Status and Fault Address registers in a PMSA implementation

This section describes the Fault Status and Fault Address registers, and how they report information about PMSA aborts. It contains the following subsections:

- *About the Fault Status and Fault Address registers*
- *Data Abort exceptions* on page B4-19
- *Prefetch Abort exceptions* on page B4-19
- *Fault Status Register encodings for the PMSA* on page B4-19
- *Distinguishing read and write accesses on Data Abort exceptions* on page B4-21
- *Provision for classification of external aborts* on page B4-21
- *Auxiliary Fault Status Registers* on page B4-21.

Also, these registers are used to report information about debug exceptions. For details see *Effects of debug exceptions on CP15 registers and the DBGWFAR* on page C4-4.

B4.5.1 About the Fault Status and Fault Address registers

PMSAv7 provides four registers for reporting fault address and status information:

- The *Data Fault Status Register*, see *c5, Data Fault Status Register (DFSR)* on page B4-55. The DFSR is updated on taking a Data Abort exception.
- The *Instruction Fault Status Register*, see *c5, Instruction Fault Status Register (IFSR)* on page B4-56. The IFSR is updated on taking a Prefetch Abort exception.
- The *Data Fault Address Register*, see *c6, Data Fault Address Register (DFAR)* on page B4-57. In some cases, on taking a synchronous Data Abort exception the DFAR is updated with the faulting address. See *Terminology for describing exceptions* on page B1-4 for a description of synchronous exceptions.
- The *Instruction Fault Address Register*, see *c6, Instruction Fault Address Register (IFAR)* on page B4-58. The IFAR is updated with the faulting address on taking a Prefetch Abort exception.

In addition, the architecture provides encodings for two IMPLEMENTATION DEFINED Auxiliary Fault Status Registers, see *Auxiliary Fault Status Registers* on page B4-21.

Note

- On a Data Abort exception that is generated by an instruction cache maintenance operation, the IFSR is also updated.
- Before ARMv7, the *Data Fault Address Register* (DFAR) was called the *Fault Address Register* (FAR).

On a Watchpoint debug exception, the *Watchpoint Fault Address Register* (DBGWFAR) is used to hold fault information. On a watchpoint access the DBGWFAR is updated with the address of the instruction that generated the Data Abort exception. For more information, see *Watchpoint Fault Address Register (DBGWFAR)* on page C10-28.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B4.5 PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ

このセクションでは、フォールトステータス レジスタとフォールトアドレス レジスタについて、およびこれらのレジスタによって PMSA アボートに関する情報がどのように報告されるかについて説明します。このセクションは、次のサブセクションで構成されています。

- *フォールトステータス レジスタとフォールトアドレス レジスタについて*
- *データアボート例外* P. B4-19
- *プリフェッチアボート例外* P. B4-19
- *PMSA のフォールトステータス レジスタのエンコード* P. B4-19
- *データアボート例外での読み出しアクセスと書き込みアクセスとの区別* P. B4-21
- *外部アボートの分類方法* P. B4-21
- *補助フォールトステータス レジスタ* P. B4-21

これらのレジスタは、デバッグ例外に関する情報の報告にも使用されます。詳細については、*CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響* P. C4-4 を参照して下さい。

B4.5.1 フォールトステータス レジスタとフォールトアドレス レジスタについて

PMSAv7 では、フォールトアドレスとステータス情報を報告するための 4 つのレジスタが提供されます。

- *データフォールト ステータスレジスタ（c5、データフォールト ステータスレジスタ (DFSR)* P. B4-55 を参照）。DFSR は、データアボート例外の取得時に更新されます。
- *命令フォールト ステータスレジスタ（c5、命令フォールト ステータスレジスタ (IFSR)* P. B4-56 を参照）。IFSR は、プリフェッチアボート例外の取得時に更新されます。
- *データフォールト アドレスレジスタ（c6、データフォールト アドレスレジスタ (DFAR)* P. B4-57 を参照）。同期データアボート例外の取得時に、DFAR がフォールトのアドレスで更新される場合があります。同期例外の説明については、*例外に関する用語* P. B1-4 を参照して下さい。
- *命令フォールト アドレスレジスタ（c6、命令フォールト アドレスレジスタ (IFAR)* P. B4-58 を参照）。IFAR は、プリフェッチアボート例外の取得時に、フォールトのアドレスで更新されます。

さらに、このアーキテクチャでは、2 つの実装定義の補助フォールト ステータスレジスタが提供されます。*補助フォールトステータス レジスタ* P. B4-21 を参照して下さい。

注

- 命令キャッシュ保守操作によって生成されたデータアボート例外で、IFSR も更新されます。
- ARMv7 以前は、*データフォールト アドレスレジスタ*(DFAR) は*フォールト アドレスレジスタ*(FAR) と呼ばれていました。

ウォッチポイントデバッグ例外では、フォールト情報を保持するために*ウォッチポイント フォールト アドレスレジスタ*(DBGWFAR) が使用されます。ウォッチポイントアクセスでは、DBGWFAR が、データアボート例外を生成した命令のアドレスで更新されます。詳細については、*ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10-28 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.5.2 Data Abort exceptions

On taking a Data Abort exception the processor:

- updates the DFSR with a fault status code
- if the Data Abort exception is synchronous:
  - — updates the DFSR with whether the faulted access was a read or a write
  - — if the Data Abort exception was not caused by a Watchpoint debug event, updates the DFAR with the address that caused the Data Abort exception
  - — if the Data Abort exception was caused by a Watchpoint debug event, the DFAR becomes UNKNOWN
- if the Data Abort exception is asynchronous, the DFAR becomes UNKNOWN.

On an access that might have multiple aborts, the MPU fault checking sequence and the prioritization of aborts determine which abort occurs. For more information, see *The MPU fault checking sequence* on page B4-15 and *Prioritization of aborts* on page B4-17.

B4.5.3 Prefetch Abort exceptions

A Prefetch Abort exception can be generated on an instruction fetch. The Prefetch Abort exception is taken synchronously with the instruction that the abort is reported on. This means:

- If the instruction is executed a Prefetch Abort exception is generated.
- If the instruction fetch is issued but the processor does not attempt to execute the instruction no Prefetch Abort exception is generated. For example, if the processor branches round the instruction no Prefetch Abort exception is generated.

On taking a Prefetch Abort exception the processor:

- updates the IFSR with a fault status code
- updates the IFAR with the address that caused the Prefetch Abort exception.

B4.5.4 Fault Status Register encodings for the PMSA

For the PMSA fault status encodings in priority order see:

- Table B4-6 for the Instruction Fault Status Register (IFSR) encodings
- Table B4-7 on page B4-20 for the Data Fault Status Register (DFSR) encodings.

Table B4-6 PMSAv7 IFSR encodings

IFSR [10,3:0][a]	Sources	IFAR	Notes
00001	Alignment fault	Valid	MPU fault
00000	Background fault	Valid	MPU fault
01101	Permission fault	Valid	MPU fault

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.5.2 データアボート例外

データアボート例外の取得時に、プロセッサは次の処理を実行します。

- フォールト ステータス コードで DFSR を更新します。
- データアボート例外が同期の場合、次の処理を行います。
  - – フォールトしたアクセスが読み出しか書き込みかで DFSR を更新する。
  - – データアボート例外を引き起こしたのがウォッチポイントデバッグ イベントでない場合は、データアボート例外を引き起こしたアドレスで DFAR を更新する。
  - – データアボート例外がウォッチポイントデバッグ イベントで引き起こされた場合、DFAR の値は未知です。
- データアボート例外が非同期の場合、DFAR の値は未知です。

複数のアボートが発生する可能性のあるアクセスでは、MPU フォールトチェック シーケンスとアボートの優先順位付けによって、どのアボートが発生するかが決定されます。詳細については、*MPU のフォールトチェック シーケンス* P. B4-15 と *アボートの優先順位付け* P. B4-17 を参照して下さい。

B4.5.3 プリフェッチアボート例外

プリフェッチアボート例外は、命令フェッチで生成される可能性があります。プリフェッチアボート例外は、アボートが報告される命令と同期して取得されます。これは次のことを意味します。

- 命令が実行されると、プリフェッチアボート例外が生成されます。
- 命令フェッチが発行されても、プロセッサがその命令の実行を試みなければ、プリフェッチアボート例外は生成されません。例えば、プロセッサがその命令の前で分岐した場合は、プリフェッチアボート例外が生成されません。

プリフェッチアボート例外の取得時に、プロセッサは次の処理を実行します。

- フォールト ステータス コードで IFSR を更新する。
- プリフェッチアボート例外を引き起こしたアドレスで IFAR を更新する。

B4.5.4 PMSA のフォールト ステータス レジスタのエンコード

優先度順の PMSA のフォールト ステータスのエンコードについては、次の表を参照して下さい。

- 表 B4-6（命令フォールト ステータスレジスタ (IFSR) のエンコード）
- P. B4-20 の表 B4-7 （データフォールト ステータスレジスタ (DFSR) のエンコード）

表 B4-6 PMSAv7 IFSR のエンコード

IFSR [10、3:0][a]	ソース	IFAR	注
00001	アライメントフォールト	有効	MPU フォールト
00000	バックグラウンドフォールト	有効	MPU フォールト
01101	アクセス許可フォールト	有効	MPU フォールト

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B4-6 PMSAv7 IFSR encodings (continued)

IFSR [10,3:0][a]	Sources	IFAR	Notes
00010	Debug event	UNKNOWN	See *Software debug events* on page C3-5
01000	Synchronous external abort	Valid	-
10100	IMPLEMENTATION DEFINED	-	Lockdown
11010	IMPLEMENTATION DEFINED	-	Coprocessor abort
11001	Memory access synchronous parity error	Valid	-

a. All IFSR[10,3:0] values not listed in this table are reserved.

Table B4-7 PMSAv7 DFSR encodings

DFSR [10,3:0][a]	Sources	DFAR	Notes
00001	Alignment fault	Valid	MPU fault
00000	Background fault	Valid	MPU fault
01101	Permission fault	Valid	MPU fault
00010	Debug event	UNKNOWN	See *Software debug events* on page C3-5
01000	Synchronous external abort	Valid	-
10100	IMPLEMENTATION DEFINED	-	Lockdown
11010	IMPLEMENTATION DEFINED	-	Coprocessor abort
11001	Memory access synchronous parity error	b	-
10110	Asynchronous external abort	UNKNOWN	-
11000	Memory access asynchronous parity error	UNKNOWN	-

a. All DFSR[10,3:0] values not listed in this table are reserved.
b. It is IMPLEMENTATION DEFINED whether the DFAR is updated for a synchronous parity error.

Note

In previous ARM documentation, the terms precise and imprecise were used instead of synchronous and asynchronous. For details of the more exact terminology introduced in this manual see *Terminology for describing exceptions* on page B1-4.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 B4-6 PMSAv7 IFSR のエンコード（続き）

IFSR [10、3:0][a]	ソース	IFAR	注
00010	デバッグイベント	未知	*ソフトウェアデバッグ イベント* P. C3-5 を参照
01000	同期外部アボート	有効	–
10100	実装定義	–	ロックダウン
11010	実装定義	–	コプロセッサアボート
11001	メモリアクセス同期パリティエラー	有効	–

a. この表に記載されていない IFSR[10、3:0] の値はすべて予約されています。

表 B4-7 PMSAv7 DFSR のエンコード

DFSR [10、3:0][a]	ソース	DFAR	注
00001	アライメントフォールト	有効	MPU フォールト
00000	バックグラウンドフォールト	有効	MPU フォールト
01101	アクセス許可フォールト	有効	MPU フォールト
00010	デバッグイベント	未知	*ソフトウェアデバッグ イベント* P. C3-5 を参照
01000	同期外部アボート	有効	–
10100	実装定義	–	ロックダウン
11010	実装定義	–	コプロセッサアボート
11001	メモリアクセス同期パリティエラー	b	–
10110	非同期外部アボート	未知	–
11000	メモリアクセス非同期パリティエラー	未知	–

a. この表に記載されていない DFSR[10、3:0] の値はすべて予約されています。
b. DFAR が同期パリティエラーについて更新されるかどうかは実装定義です。

注

以前の ARM 説明書では、同期と非同期の代わりに、正確と不正確という用語が使用されていました。本書で採用された用語の詳細については、*例外に関する用語* P. B1-4 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Reserved encodings in the IFSR and DFSR encodings tables

A single encoding is reserved for cache lockdown faults. The details of these faults and any associated subsidiary registers are IMPLEMENTATION DEFINED.

A single encoding is reserved for aborts associated with coprocessors. The details of these faults are IMPLEMENTATION DEFINED.

B4.5.5 Distinguishing read and write accesses on Data Abort exceptions

On a Data Abort exception, the DFSR.WnR bit, bit [11] of the register, indicates whether the abort occurred on a read access or on a write access. However, for a fault on a CP15 cache maintenance operation this bit always indicates a write access fault.

For a fault generated by a `SWP` or `SWPB` instruction, the WnR bit is 0 if a read to the location would have generated a fault, otherwise it is 1.

B4.5.6 Provision for classification of external aborts

An implementation can use the DFSR.ExT and IFSR.ExT bits to provide more information about external aborts:

- DFSR.ExT can provide an IMPLEMENTATION DEFINED classification of external aborts on data accesses
- IFSR.ExT can provide an IMPLEMENTATION DEFINED classification of external aborts on instruction accesses

For all aborts other than external aborts these bits return a value of 0.

B4.5.7 Auxiliary Fault Status Registers

ARMv7 architects two Auxiliary Fault Status Registers:

- the Auxiliary Data Fault Status Register (ADFSR)
- the Auxiliary Instruction Fault Status Register (AIFSR).

These registers enable additional fault status information to be returned:

- The position of these registers is architecturally-defined, but the content and use of the registers is IMPLEMENTATION DEFINED.
- An implementation that does not need to report additional fault information must implement these registers as UNK/SBZ. This ensures that a privileged attempt to access these registers is not faulted.

An example use of these registers would be to return more information for diagnosing parity errors.

See *c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B4-56 for the architectural details of these registers.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

IFSR と DFSR のエンコード表で予約済みのエンコード

1 つのエンコードが、キャッシュロックダウン フォールト用に予約されています。これらのフォールトおよび関連する補助レジスタの詳細は、実装定義です。

1 つのエンコードが、コプロセッサに関連付けられたアボート用に予約されています。これらのフォールトの詳細は、実装定義です。

B4.5.5 データアボート例外での読み出しアクセスと書き込みアクセスとの区別

データアボート例外では、レジスタの DFSR.WnR ビット（ビット [11]）によって、アボートが読み出しアクセスと書き込みアクセスのどちらで発生したかが示されます。ただし、CP15 キャッシュ保守操作でのフォールトの場合は、このビットは常に書き込みアクセスフォールトを示します。

`SWP` 命令または `SWPB` 命令によって生成されたフォールトの場合は、その位置に対する読み出しによってフォールトが生成された場合は WnR ビットは 0、それ以外の場合は WnR ビットは 1 です。

B4.5.6 外部アボートの分類方法

実装で、DFSR.ExT ビットと IFSR.ExT ビットを使用して、外部アボートの詳細情報を提供することができます。

- DFSR.ExT を使用すれば、データアクセスでの外部アボートについて実装定義の分類が可能になります。
- IFSR.ExT を使用すれば、命令アクセスでの外部アボートについて実装定義の分類が可能になります。

外部アボート以外のアボートでは、これらのビットは 0 の値を返します。

B4.5.7 補助フォールトステータス レジスタ

ARMv7 では、次の 2 つの補助フォールトステータス レジスタが追加されました。

- 補助データフォールト ステータスレジスタ (ADFSR)
- 補助命令フォールト ステータスレジスタ (AIFSR)

これらのレジスタを使用すれば、追加のフォールトステータス情報を返すことができます。

- これらのレジスタの位置付けはアーキテクチャ定義ですが、レジスタの内容と用途は実装定義です。
- 追加のフォールト情報を報告する必要のない実装では、これらのレジスタを UNK/SBZ として実装する必要があります。これによって、これらのレジスタに対する特権アクセスがフォールトしないことが保証されます。

これらのレジスタの使用例としては、パリティエラーの診断に関する詳細情報の提供があります。

これらのレジスタのアーキテクチャ上の詳細については、*c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B4- 56 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6 CP15 registers for a PMSA implementation

This section gives a full description of the registers implemented in the CP15 System Control Coprocessor in an ARMv7 implementation that includes the PMSA memory system. Therefore, this is the description of the CP15 registers for an ARMv7-R implementation.

Some of the registers described in this section are also included in an ARMv7 implementation with a VMSA. The section *CP15 registers for a VMSA implementation* on page B3-64 also includes descriptions of these registers.

See *Coprocessors and system control* on page B1-62 for general information about the System Control Coprocessor, CP15 and the register access instructions `MRC` and `MCR`.

Information in this section is organized as follows:

- general information is given in:
  - — *Organization of the CP15 registers in a PMSA implementation*
  - — *General behavior of CP15 registers* on page B4-26
  - — *Changes to CP15 registers and the memory order model* on page B4-28
  - — *Meaning of fixed bit values in register diagrams* on page B4-29.
- this is followed by, for each of the primary CP15 registers c0 to c15:
  - — a general description of the organization of the primary CP15 register
  - — detailed descriptions of all the registers in that primary register.

Note

The detailed descriptions of the registers that implement the processor identification scheme, CPUID, are given in Chapter B5 *The CPUID Identification Scheme*, and not in this section.

Table B4-8 on page B4-24 lists all of the CP15 registers in a PMSA implementation, and is an index to the detailed description of each register.

B4.6.1 Organization of the CP15 registers in a PMSA implementation

Figure B4-3 on page B4-23 summarizes the ARMv7 CP15 registers when the PMSA is implemented. Table B4-8 on page B4-24 lists all of these registers.

Note

ARMv7 introduces significant changes to the memory system registers, especially in relation to caches. For details of:

- the CP15 register implementation in PMSAv6, see *Organization of CP15 registers for an ARMv6 PMSA implementation* on page AppxG-31.
- how the ARMv7 registers must be used to discover what caches can be accessed by the processor, see *Identifying the cache resources in ARMv7* on page B2-4.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B4.6 PMSA 実装の CP15 レジスタ

このセクションでは、PMSA メモリシステムを含む ARMv7 実装の CP15 システム制御コプロセッサに実装されるレジスタについて詳しく説明します。つまり、これは ARMv7-R 実装の CP15 レジスタについての説明です。

このセクションで説明するレジスタの一部は、VMSA を含む ARMv7 実装にも存在します。これらのレジスタについては、*VMSA 実装の CP15 レジスタ* P. B3-64 でも説明されています。

システム制御コプロセッサ CP15 と、レジスタアクセス命令 `MRC` および `MCR` の一般的な情報については、*コプロセッサとシステム制御* P. B1-62 を参照して下さい。

このセクションの情報は、次のように編成されています。

- 一般的な情報については、次のサブセクションで説明します。
  - – *PMSA 実装の CP15 レジスタの編成*
  - – *CP15 レジスタの一般的な動作* P. B4-26
  - – *CP15 レジスタの変更とメモリオーダ モデル* P. B4-28
  - – *レジスタ図の固定ビット値の意味* P. B4-29
- その後で、1 次 CP15 レジスタの c0 ～ c15 のそれぞれについて次のサブセクションで説明します。
  - – 1 次 CP15 レジスタの編成についての一般的な説明
  - – その 1 次レジスタに含まれるすべてのレジスタについての詳細な説明

注

プロセッサ識別方式の CPUID が実装されるレジスタの詳細については、このセクションではなく、第 B5 章 *「CPUID 識別方式」* で説明します。

PMSA 実装のすべての CP15 レジスタと各レジスタの詳細への索引を、P. B4-24 の表 B4-8 に示します。

B4.6.1 PMSA 実装の CP15 レジスタの編成

P. B4-23 の図 B4-3 は、PMSA が実装されている場合の ARMv7 CP15 レジスタをまとめたものです。これらのレジスタの一覧を、P. B4-24 の表 B4-8 に示します。

注

ARMv7 では、メモリシステム レジスタ、特にキャッシュ関連のレジスタに大幅な変更が加えられています。詳細については、次のセクションを参照して下さい。

- PMSAv6 の CP15 レジスタの実装（*ARMv6 PMSA 実装での CP15 レジスタの編成* P. 付録 G-31 を参照）
- ARMv7 レジスタを使用して、プロセッサからアクセス可能なキャッシュを特定する方法（*ARMv7 におけるキャッシュリソースの識別* P. B2-4 を参照）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure B4-3 CP15 registers in a PMSA implementation

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

図 B4- 3 PMSA 実装の CP15 レジスタ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For information about the CP15 encodings not shown in Figure B4-3 on page B4-23 see *Unpredictable and undefined behavior for CP15 accesses* on page B4-26.

Summary of CP15 register descriptions in a PMSA implementation

Table B4-8 shows the CP15 registers in a PMSA implementation. The table also includes links to the descriptions of each of the primary CP15 registers, c0 to c15.

Table B4-8 Summary of CP15 registers in a PMSA implementation

Register and description
CP15 c0, ID codes registers on page B4-30
c0, Main ID Register (MIDR) on page B4-32
c0, Cache Type Register (CTR) on page B4-34
c0, TCM Type Register (TCMTR) on page B4-35
c0, MPU Type Register (MPUIR) on page B4-36
c0, Multiprocessor Affinity Register (MPIDR) on page B4-37
CP15 c0, Processor Feature registers on page B5-4
c0, Debug Feature Register 0 (ID_DFR0) on page B5-6
c0, Auxiliary Feature Register 0 (ID_AFR0) on page B5-8
CP15 c0, Memory Model Feature registers on page B5-9
CP15 c0, Instruction Set Attribute registers on page B5-19
c0, Cache Size ID Registers (CCSIDR) on page B4-40
c0, Cache Level ID Register (CLIDR) on page B4-41
c0, Implementation defined Auxiliary ID Register (AIDR) on page B4-43
c0, Cache Size Selection Register (CSSELR) on page B4-43
CP15 c1, System control registers on page B4-44
c1, System Control Register (SCTLR) on page B4-45
c1, Implementation defined Auxiliary Control Register (ACTLR) on page B4-50
c1, Coprocessor Access Control Register (CPACR) on page B4-51
CP15 registers c2, c3, and c4 are not used on a PMSA implementation, see *Unallocated CP15 encodings* on page B4-27

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

P. B4-23 の図 B4-3 に示されていない CP15 エンコードの詳細については、*CP15 アクセスの予測不能動作と未定義動作* P. B4-26 を参照して下さい。

PMSA 実装の CP15 レジスタの概要

PMSA 実装内の CP15 レジスタを、表 B4-8 に示します。この表には、1 次 CP15 レジスタの c0 ～ c15 のそれぞれに関する説明へのリンクも含まれています。

表 B4-8 PMSA 実装の CP15 レジスタの概要

レジスタと説明
CP15 c0、ID コードレジスタ P. B4-30
c0、メイン ID レジスタ (MIDR) P. B4-32
c0、キャッシュタイプ レジスタ (CTR) P. B4-34
c0、TCM タイプレジスタ (TCMTR) P. B4-35
c0、MPU タイプレジスタ (MPUIR) P. B4-36
c0、マルチプロセッサ類似性レジスタ (MPIDR) P. B4-37
CP15 c0、プロセッサ機能レジスタ P. B5-4
c0、デバッグ機能レジスタ 0 (ID_DFR0) P. B5-6
c0、補助機能レジスタ 0 (ID_AFR0) P. B5-8
CP15 c0、メモリモデル機能レジスタ P. B5-9
CP15 c0、命令セット属性レジスタ P. B5-19
c0、キャッシュサイズ ID レジスタ (CCSIDR) P. B4-40
c0、キャッシュレベル ID レジスタ (CLIDR) P. B4-41
c0、実装定義の補助 ID レジスタ (AIDR) P. B4-43
c0、キャッシュサイズ選択レジスタ (CSSELR) P. B4-43
CP15 c1、システム制御レジスタ P. B4-44
c1、システム制御レジスタ (SCTLR) P. B4-45
c1、実装定義の補助制御レジスタ (ACTLR) P. B4-50
c1、コプロセッサアクセス制御レジスタ (CPACR) P. B4-51
CP15 レジスタの c2、c3、c4 は、PMSA 実装では使用されません（*未割り当ての CP15 エンコード* P. B4-27 を参照）。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B4-8 Summary of CP15 registers in a PMSA implementation (continued)

Register and description
CP15 c5 and c6, Memory system fault registers on page B4-53
c5, Data Fault Status Register (DFSR) on page B4-55
c5, Instruction Fault Status Register (IFSR) on page B4-56
c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR) on page B4-56
c6, Data Fault Address Register (DFAR) on page B4-57
c6, Instruction Fault Address Register (IFAR) on page B4-58
c6, Data Region Base Address Register (DRBAR) on page B4-60
c6, Instruction Region Base Address Register (IRBAR) on page B4-61
c6, Data Region Size and Enable Register (DRSR) on page B4-62
c6, Instruction Region Size and Enable Register (IRSR) on page B4-63
c6, Data Region Access Control Register (DRACR) on page B4-64
c6, Instruction Region Access Control Register (IRACR) on page B4-65
c6, MPU Region Number Register (RGNR) on page B4-66
CP15 c7, Cache maintenance and other functions on page B4-68
CP15 c7, Cache and branch predictor maintenance functions on page B4-68
CP15 c7, Data and Instruction Barrier operations on page B4-72
CP15 c7, No Operation (NOP) on page B4-73
CP15 c8 is not used on a PMSA implementation, see *Unallocated CP15 encodings* on page B4-27
CP15 c9, Cache and TCM lockdown registers and performance monitors on page B4-74
CP15 c10 is not used on a PMSA implementation, see *Unallocated CP15 encodings* on page B4-27
CP15 c11, Reserved for TCM DMA registers on page B4-75
CP15 c12 is not used on a PMSA implementation, see *Unallocated CP15 encodings* on page B4-27

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B4-8 PMSA 実装の CP15 レジスタの概要（続き）

レジスタと説明
CP15 c5 および c6、メモリシステム フォールトレジスタ P. B4-53
c5、データフォールト ステータスレジスタ (DFSR) P. B4-55
c5、命令フォールト ステータスレジスタ (IFSR) P. B4-56
c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR) P. B4-56
c6、データフォールト アドレスレジスタ (DFAR) P. B4-57
c6、命令フォールト アドレスレジスタ (IFAR) P. B4-58
c6、データ領域ベースアドレス レジスタ (DRBAR) P. B4-60
c6、命令領域ベースアドレス レジスタ (IRBAR) P. B4-61
c6、データ領域サイズおよびイネーブルレジスタ (DRSR) P. B4-62
c6、命令領域サイズおよびイネーブルレジスタ (IRSR) P. B4-63
c6、データ領域アクセス制御レジスタ (DRACR) P. B4-64
c6、命令領域アクセス制御レジスタ (IRACR) P. B4-65
c6、MPU 領域番号レジスタ (RGNR) P. B4-66
CP15 c7、キャッシュ保守機能とその他の機能 P. B4-68
CP15 c7、キャッシュおよび分岐予測器保守機能 P. B4-68
CP15 c7、データおよび命令バリア操作 P. B4-72
CP15 c7、無操作 (NOP) P. B4-73
PMSA 実装では CP15 c8 は使用されません（*未割り当ての CP15 エンコード* P. B4-27 を参照）。
CP15 c9、キャッシュおよび TCM ロックダウンレジスタとパフォーマンスモニタ P. B4-74
PMSA 実装では CP15 c10 は使用されません（*未割り当ての CP15 エンコード* P. B4-27 を参照）。
CP15 c11、TCM DMA 用に予約されているレジスタ P. B4-75
PMSA 実装では CP15 c12 は使用されません（*未割り当ての CP15 エンコード* P. B4-27 を参照）。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B4-8 Summary of CP15 registers in a PMSA implementation (continued)

Register and description
CP15 c13, Context and Thread ID registers on page B4-75
c13, Context ID Register (CONTEXTIDR) on page B4-76
CP15 c13 Software Thread ID registers on page B4-77
CP15 c14 is not used on a PMSA implementation, see *Unallocated CP15 encodings* on page B4-27
CP15 c15, Implementation defined registers on page B4-78

B4.6.2 General behavior of CP15 registers

The following sections give information about the general behavior of CP15 registers:

- *Unpredictable and undefined behavior for CP15 accesses*
- *Reset behavior of CP15 registers* on page B4-27

See also *Meaning of fixed bit values in register diagrams* on page B4-29.

Read-only bits in read/write registers

Some read/write registers include bits that are read-only. These bits ignore writes.

An example of this is the SCTLR.NMFI bit, bit [27], see *c1, System Control Register (SCTLR)* on page B4-45.

UNPREDICTABLE and UNDEFINED behavior for CP15 accesses

In ARMv7 the following operations are UNDEFINED:

- all `CDP`, `MCRR`, `MRRC`, `LDC` and `STC` operations to CP15
- all `CDP2`, `MCR2`, `MRC2`, `MCRR2`, `MRRC2`, `LDC2` and `STC2` operations to CP15.

Unless otherwise indicated in the individual register descriptions:

- reserved fields in registers are UNK/SBZP
- reserved values of fields can have UNPREDICTABLE effects.

The following subsections give more information about UNPREDICTABLE and UNDEFINED behavior for CP15:

- *Unallocated CP15 encodings* on page B4-27
- *Rules for MCR and MRC accesses to CP15 registers* on page B4-27.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 B4-8 PMSA 実装の CP15 レジスタの概要（続き）

レジスタと説明
CP15 c13、コンテキストおよびスレッド ID レジスタ P. B4-75
c13、コンテキスト ID レジスタ (CONTEXTIDR) P. B4-76
CP15 c13、ソフトウェアスレッド ID レジスタ P. B4-77
PMSA 実装では CP15 c14 は使用されません（*未割り当ての CP15 エンコード* P. B4-27 を参照）。
CP15 c15、実装定義のレジスタ P. B4-78

B4.6.2 CP15 レジスタの一般的な動作

次のセクションでは、CP15 レジスタの一般的な動作について説明します。

- *CP15 アクセスの予測不能動作と未定義動作*
- *CP15 レジスタのリセット時の動作* P. B4-27

レジスタ図の固定ビット値の意味 P. B4-29 も参照して下さい。

読み出し / 書き込みレジスタの読み出し専用ビット

一部の読み出し / 書き込みレジスタには、読み出し専用のビットがあります。これらのビットへの書き込みは無視されます。

例としては、SCTLR.NMFI ビット（ビット [27]）が挙げられます。*c1、システム制御レジスタ (SCTLR)* P. B4-45 を参照して下さい。

CP15 アクセスの予測不能動作と未定義動作

ARMv7 では、次の操作が未定義です。

- CP15 に対するすべての `CDP`、`MCRR`、`MRRC`、`LDC`、`STC` 操作
- CP15 に対するすべての `CDP2`、`MCR2`、`MRC2`、`MCRR2`、`MRRC2`、`LDC2`、`STC2` 操作

個別のレジスタの説明の中で特に記載がない場合、次の規則が適用されます。

- レジスタの予約済みフィールドは UNK/SBZP です。
- フィールドの予約済みの値は、予測不能な影響を及ぼすことがあります。

次のサブセクションでは、CP15 の予測不能動作と未定義動作についてさらに詳しく説明します。

- *未割り当ての CP15 エンコード* P. B4-27
- *CP15 レジスタへの MCR および MRC アクセスに関する規則* P. B4-27

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Unallocated CP15 encodings

When MCR and MRC instructions perform CP15 operations, the <CRn> value for the instruction is the major register specifier for the CP15 space. Accesses to unallocated major registers are UNDEFINED. For the ARMv7-R Architecture, this means that accesses with <CRn> = {c2-c4, c8, c10, c12, c14} are UNDEFINED.

In an allocated CP15 major register specifier, MCR and MRC accesses to all unallocated encodings are UNPREDICTABLE for privileged accesses. For the ARMv7-A architecture this means that privileged MCR and MRC accesses with <CRn> != {c2-c4, c8, c10, c12, c14} but with an unallocated combination of <opc1>, <CRm> and <opc2> values, are UNPREDICTABLE. For <CRn> != {c2-c4, c8, c10, c12, c14}, Figure B4-3 on page B4-23 shows all allocated allocations of <opc1>, <CRm> and <opc2>. A privileged access using any combination not show in the figure is UNPREDICTABLE.

Note

As shown in Figure B4-3 on page B4-23, accesses to unallocated principal ID registers map onto the Main ID Register. These are accesses with <CRn> = c0, <opc1> = 0, <CRm> = c0, and <opc2> = {4, 6, 7}.

Rules for MCR and MRC accesses to CP15 registers

All MCR operations from the PC are UNPREDICTABLE for all coprocessors, including for CP15.

All MRC operations to APSR_nzcv are UNPREDICTABLE for CP15.

The following accesses are UNPREDICTABLE:

- an MCR access to an encoding for which no write behavior is defined in any circumstances
- an MRC access to an encoding for which no read behavior is defined in any circumstances.

Except for CP15 encoding that are accessible in User mode, all MCR and MRC accesses from User mode are UNDEFINED. This applies to all User mode accesses to unallocated CP15 encodings. Individual register descriptions, and the summaries of the CP15 major registers, show the CP15 encodings that are accessible in User mode.

Some individual registers can be made inaccessible by setting configuration bits, possibly including IMPLEMENTATION DEFINED configuration bits, to disable access to the register. The effects of the architecturally-defined configuration bits are defined individually in this manual. Typically, setting a configuration bit to disable access to a register results in the register becoming UNDEFINED for MRC and MCR accesses.

Reset behavior of CP15 registers

After a reset, only a limited subset of the processor state is guaranteed to be set to defined values. On reset, the PMSAv7 architecture requires that the following CP15 registers are set to defined values:

- the SCTLR, see *c1, System Control Register (SCTLR)* on page B4-45
- the CPACR, see *c1, Coprocessor Access Control Register (CPACR)* on page B4-51
- the DRSR, see *c6, Data Region Size and Enable Register (DRSR)* on page B4-62
- the IRSR, if implemented, see *c6, Instruction Region Size and Enable Register (IRSR)* on page B4-63.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

未割り当ての CP15 エンコード

MCR 命令と MRC 命令で CP15 操作を実行する場合は、命令の <CRn> 値が CP15 空間の主要レジスタ指定子です。未割り当ての主要レジスタへのアクセスは未定義です。ARMv7-R アーキテクチャの場合、これは <CRn> = {c2-c4, c8, c10, c12, c14} を使用したアクセスが未定義になることを意味します。

割り当て済みの CP15 主要レジスタ指定子では、未割り当てのエンコードに対する MCR と MRC のアクセスはすべて、特権アクセスの場合に予測不能です。ARMv7-A アーキテクチャの場合、これは <CRn> != {c2-c4, c8, c10, c12, c14} であるが、<opc1>、<CRm>、<opc2> の値の未割り当ての組み合わせを使用した、MCR または MRC の特権アクセスが予測不能になることを意味します。<CRn> != {c2-c4, c8, c10, c12, c14} の場合の <opc1>、<CRm>、<opc2> のすべての割り当てを、P. B4-23 の図 B4-3 に示します。この図に示されていない組み合わせを使用した特権アクセスは、予測不能です。

注

P. B4-23 の図 B4-3 に示すように、未割り当ての基本 ID レジスタへのアクセスは、メイン ID レジスタにマッピングされます。これらのアクセスは、<CRn> = c0、<opc1> = 0、<CRm> = c0、<opc2> = {4, 6, 7} を使用したアクセスです。

CP15 レジスタへの MCR および MRC アクセスに関する規則

PC からのすべての MCR 操作は、CP15 を含むすべてのコプロセッサについて予測不能です。

APSR_nzcv へのすべての MRC 操作は、CP15 について予測不能です。

次のアクセスは予測不能です。

- いずれかの状況について書き込み動作が定義されていないエンコードへの MCR アクセス
- いずれかの状況について読み出し動作が定義されていないエンコードへの MRC アクセス

ユーザモードでアクセス可能な CP15 エンコードを除いて、ユーザモードからのすべての MCR および MRC アクセスは未定義です。このことは、未割り当ての CP15 エンコードへのユーザモード アクセスにも当てはまります。個々のレジスタの説明と CP15 主要レジスタの概要には、ユーザモードでアクセス可能な CP15 エンコードが示されています。

一部のレジスタは、レジスタへのアクセスを禁止にするように構成ビット（実装定義の構成ビットが含まれる場合があります）を設定することによって、アクセス不可にすることができます。アーキテクチャ定義の構成ビットの影響は、本書内で個別に定義されます。一般的に、レジスタへのアクセスを禁止するように構成ビットを設定すると、レジスタが MRC および MCR アクセスについて未定義になります。

CP15 レジスタのリセット時の動作

リセット後は、プロセッサ状態の制限されたサブセットのみが、定義された値に設定されることが保証されます。PMSAv7 アーキテクチャでは、リセット時に、次の CP15 レジスタを定義された値に設定する必要があります。

- SCTLR（*c1、システム制御レジスタ (SCTLR)* P. B4-45 を参照）
- CPACR（*c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B4-51 を参照）
- DRSR（*c6、データ領域サイズおよびイネーブルレジスタ (DRSR)* P. B4-62 を参照）
- 実装されている場合、IRSR（*c6、命令領域サイズおよびイネーブルレジスタ (IRSR)* P. B4-63 を参照）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For details of the reset values of these registers see the register descriptions.

After a reset, software must not rely on the value of any read/write register not included in this list.

B4.6.3 Changes to CP15 registers and the memory order model

All changes to CP15 registers that appear in program order after any explicit memory operations are guaranteed not to affect those memory operations.

Any change to CP15 registers is guaranteed to be visible to subsequent instructions only after one of:

- the execution of an ISB instruction
- the taking of an exception
- the return from an exception.

To guarantee the visibility of changes to some CP15 registers, additional operations might be required, on a case by case basis, before the ISB instruction, exception or return from exception. These cases are identified specifically in the definition of the registers.

However, for CP15 register accesses, all MRC and MCR instructions to the same register using the same register number appear to occur in program order relative to each other without context synchronization.

Where a change to the CP15 registers that is not yet guaranteed to be visible has an effect on exception processing, the following rule applies:

- When it is determined that an exception must be taken, any change of state held in CP15 registers involved in the triggering of the exception and that affects the processing of the exception is guaranteed to take effect before the exception is taken.

Therefore, in the following example, where initially A=1 and V=0, the LDR might or might not take a Data Abort exception due to the unaligned access, but if an exception occurs, the vector used is affected by the V bit:

```
MCR p15, R0, c1, c0, 0     ; clears the A bit and sets the V bit
LDR R2, [R3]               ; unaligned load.
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

これらのレジスタにおけるリセット時の値の詳細については、各レジスタの説明を参照して下さい。

リセット後は、このリストに含まれていない読み出し / 書き込みレジスタの値について、ソフトウェアが特定の値を前提とすることはできません。

B4.6.3 CP15 レジスタの変更とメモリオーダ モデル

プログラムの順序で明示的なメモリ操作の後に出現する CP15 レジスタの変更は、それらのメモリ操作に影響を与えないことが保証されます。

CP15 レジスタの変更は、次のいずれかの後でのみ、以後の命令から可視であることが保証されます。

- ISB 命令の実行
- 例外の取得
- 例外からの復帰

一部の CP15 レジスタの変更の可視性を保証するには、ISB 命令、例外、または例外からの復帰の前に、条件に応じて追加の操作が必要な場合があります。このような条件は、レジスタの定義で明確に特定されます。

ただし、CP15 レジスタへのアクセスでは、同じレジスタ番号を使用した同じレジスタへの MRC 命令と MCR 命令のすべてが、相対的にプログラムでの順序で、コンテキスト同期なしに発生するように見えます。

可視であることが保証されていない CP15 レジスタの変更が例外処理に影響を与える場合、次の規則が適用されます。

- 例外を取得する必要があることが判明した場合、CP15 レジスタに保持されている状態の変更のうち、例外のトリガに関係したものと、例外の処理に影響を与えるものは、その例外が取得される前に反映されることが保証されます。

したがって、次の例では、初期状態で A = 1 および V = 0 の場合は、LDR によって、アンアラインドアクセスが原因のデータアボート例外が取得される場合と取得されない場合がありますが、例外が発生した場合は、使用されるベクタが V ビットの影響を受けます。

```
MCR p15, R0, c1, c0, 0     ; clears the A bit and sets the V bit
LDR R2, [R3]               ; unaligned load
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.4 Meaning of fixed bit values in register diagrams

In register diagrams, fixed bits are indicated by one of following:

0 In any implementation:
- the bit must read as 0
- writes to the bit must be ignored.

Software:
- can rely on the bit reading as 0
- must use an SBZP policy to write to the bit.

(0) In any implementation:
- the bit must read as 0
- writes to the bit must be ignored.

Software:
- must not rely on the bit reading as 0
- must use an SBZP policy to write to the bit.

1 In any implementation:
- the bit must read as 1
- writes to the bit must be ignored.

Software:
- can rely on the bit reading as 1
- must use an SBOP policy to write to the bit.

(1) In any implementation:
- the bit must read as 1
- writes to the bit must be ignored.

Software:
- must not rely on the bit reading as 1
- must use an SBOP policy to write to the bit.

Fields that are more than 1 bit wide are sometimes described as UNK/SBZP, instead of having each bit marked as (0).

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.4 レジスタ図の固定ビット値の意味

レジスタ図では、固定ビットは次のいずれかによって示されます。

0 任意の実装において
- ビットは 0 として読み出される必要がある。
- ビットへの書き込みは無視される必要がある。

ソフトウェアでは
- ビットが 0 として読み出されることを前提とできる。
- SBZP ポリシーを使用してビットに書き込む必要がある。

(0) 任意の実装において
- ビットは 0 として読み出される必要がある。
- ビットへの書き込みは無視される必要がある。

ソフトウェアでは
- ビットが 0 として読み出されることを前提とできない。
- SBZP ポリシーを使用してビットに書き込む必要がある。

1 任意の実装において
- ビットは 1 として読み出される必要がある。
- ビットへの書き込みは無視される必要がある。

ソフトウェアでは
- ビットが 1 として読み出されることを前提とできる。
- SBOP ポリシーを使用してビットに書き込む必要がある。

(1) 任意の実装において
- ビットは 1 として読み出される必要がある。
- ビットへの書き込みは無視される必要がある。

ソフトウェアでは
- ビットが 1 として読み出されることを前提とできない。
- SBOP ポリシーを使用してビットに書き込む必要がある。

1 ビット幅を超えるフィールドは、各ビットを (0) としてマークする代わりに、UNK/SBZP として表記される場合があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.5 CP15 c0, ID codes registers

The CP15 c0 registers are used for processor and feature identification. Figure B4-4 shows the CP15 c0 registers.

Figure B4-4 CP15 c0 registers in a PMSA implementation

All CP15 c0 register encodings not shown in Figure B4-4 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B4-27.

Note

Chapter B5 *The CPUID Identification Scheme* describes the CPUID registers shown in Figure B4-4.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B4.6.5 CP15 c0、ID コードレジスタ

CP15 c0 レジスタは、プロセッサと機能の識別に使用されます。CP15 c0 レジスタを、図 B4-4 に示します。

図 B4-4 PMSA 実装の CP15 c0 レジスタ

図 B4-4 に示されていない CP15 c0 レジスタのすべてのエンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

注

図 B4-4 に示されている CPUID レジスタについては、第 B5 章 *「CPUID 識別方式」*で説明します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B4-9 lists the CP15 c0 registers and shows where each register is described in full. The table does not include the reserved and aliased registers that are shown in Figure B4-4 on page B4-30.

Table B4-9 Index to CP15 c0 register descriptions

opc1	CRm	opc2	Register and description
0	c0	0	*c0, Main ID Register (MIDR)* on page B4-32
		1	*c0, Cache Type Register (CTR)* on page B4-34
		2	*c0, TCM Type Register (TCMTR)* on page B4-35
		4	*c0, MPU Type Register (MPUIR)* on page B4-36
		5	*c0, Multiprocessor Affinity Register (MPIDR)* on page B4-37
		3, 6, 7	*c0, Main ID Register (MIDR)* on page B4-32
	c1	0, 1	*CP15 c0, Processor Feature registers* on page B5-4
		2	*c0, Debug Feature Register 0 (ID_DFR0)* on page B5-6
		3	*c0, Auxiliary Feature Register 0 (ID_AFR0)* on page B5-8
		4-7	*CP15 c0, Memory Model Feature registers* on page B5-9
	c2	0-5	*CP15 c0, Instruction Set Attribute registers* on page B5-19
1	c0	0	*c0, Cache Size ID Registers (CCSIDR)* on page B4-40
		1	*c0, Cache Level ID Register (CLIDR)* on page B4-41
		7	*c0, Implementation defined Auxiliary ID Register (AIDR)* on page B4-43
2	c0	0	*c0, Cache Size Selection Register (CSSELR)* on page B4-43

Note

The CPUID scheme described in Chapter B5 *The CPUID Identification Scheme* includes information about the implementation of the optional Floating-Point and Advanced SIMD architecture extensions. See *Advanced SIMD and VFP extensions* on page A2-20 for a summary of the implementation options for these features.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP15 c0 レジスタと、各レジスタについて完全な説明のあるセクションを、表 B4-9 に示します。この表には、P. B4-30 の図 B4-4 に示す予約済みのレジスタとエイリアスレジスタは含まれていません。

表 B4-9 CP15 c0 レジスタの説明の索引

opc1	CRm	opc2	レジスタと説明
0	c0	0	*c0、メイン ID レジスタ (MIDR)* P. B4-32
		1	*c0、キャッシュタイプ レジスタ (CTR)* P. B4-34
		2	*c0、TCM タイプレジスタ (TCMTR)* P. B4-35
		4	*c0、MPU タイプレジスタ (MPUIR)* P. B4-36
		5	*c0、マルチプロセッサ類似性レジスタ (MPIDR)* P. B4-37
		3、6、7	*c0、メイン ID レジスタ (MIDR)* P. B4-32
	c1	0、1	*CP15 c0、プロセッサ機能レジスタ* P. B5-4
		2	*c0、デバッグ機能レジスタ 0 (ID_DFR0)* P. B5-6
		3	*c0、補助機能レジスタ 0 (ID_AFR0)* P. B5-8
		4–7	*CP15 c0、メモリモデル機能レジスタ* P. B5-9
	c2	0–5	*CP15 c0、命令セット属性レジスタ* P. B5-19
1	c0	0	*c0、キャッシュサイズ ID レジスタ (CCSIDR)* P. B4-40
		1	*c0、キャッシュレベル ID レジスタ (CLIDR)* P. B4-41
		7	*c0、実装定義の補助 ID レジスタ (AIDR)* P. B4-43
2	c0	0	*c0、キャッシュサイズ選択レジスタ (CSSELR)* P. B4-43

注

第 B5 章 *「CPUID 識別方式」* で説明されている CPUID 方式には、オプションの浮動小数点およびアドバンスト SIMD アーキテクチャ拡張機能の実装に関する情報が含まれています。これらの機能の実装オプションの概要については、*アドバンスト SIMD および VFP 拡張機能* P. A2-20 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.6 c0, Main ID Register (MIDR)

The Main ID Register, MIDR, provides identification information for the processor, including an implementer code for the device and a device ID number.

The MIDR is:

- a 32-bit read-only register
- accessible only in privileged modes.

Some fields of the MIDR are IMPLEMENTATION DEFINED. For details of the values of these fields for a particular ARMv7 implementation, and any implementation-specific significance of these values, see the product documentation.

The format of the MIDR is:

31 – 24	23 – 20	19 – 16	15 – 4	3 – 0
Implementer	Variant	Architecture	Primary part number	Revision

Implementer, bits [31:24]

The Implementer code. Table B4-10 shows the permitted values for this field:

Table B4-10 Implementer codes

Bits [31:24]	ASCII character	Implementer
0x41	A	ARM Limited
0x44	D	Digital Equipment Corporation
0x4D	M	Motorola, Freescale Semiconductor Inc.
0x51	Q	QUALCOMM Inc.
0x56	V	Marvell Semiconductor Inc.
0x69	i	Intel Corporation

All other values are reserved by ARM and must not be used.

Variant, bits [23:20]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, for example implementations of the same product with different cache sizes.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B4.6.6 c0、メイン ID レジスタ (MIDR)

メイン ID レジスタ (MIDR) では、デバイスの実装者コードやデバイス ID 番号を含む、プロセッサの識別情報が提供されます。

MIDR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能

MIDR の一部のフィールドは、実装定義です。特定の ARMv7 実装におけるこれらのフィールドの値と、これらの値の実装固有の意味については、製品説明書を参照して下さい。

MIDR のフォーマットは次のとおりです。

31 – 24	23 – 20	19 – 16	15 – 4	3 – 0
実装者	バリアント	アーキテクチャ	主要部品番号	リビジョン

実装者、ビット [31:24]

実装者コード。このフィールドに許可される値を、表 B4-10 に示します。

表 B4-10 実装者コード

ビット [31:24]	ASCII 文字	実装者
0x41	A	ARM Limited
0x44	D	Digital Equipment Corporation
0x4D	M	Motorola、Freescale Semiconductor Inc.
0x51	Q	QUALCOMM Inc.
0x56	V	Marvell Semiconductor Inc.
0x69	i	Intel Corporation

他のすべての値は ARM によって予約されており、使用できません。

バリアント、ビット [23:20]

実装定義のバリアント番号。一般的に、このフィールドは、キャッシュサイズが異なる同じ製品の実装などの、複数の製品バリアントを区別するために使用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Architecture, bits [19:16]

Table B4-11 shows the permitted values for this field:

Table B4-11 Architecture codes

Bits [19:16]	Architecture
0x1	ARMv4
0x2	ARMv4T
0x3	ARMv5 (obsolete)
0x4	ARMv5T
0x5	ARMv5TE
0x6	ARMv5TEJ
0x7	ARMv6
0xF	Defined by CPUID scheme

All other values are reserved by ARM and must not be used.

Primary part number, bits [15:4]

An IMPLEMENTATION DEFINED primary part number for the device.

Note

On processors implemented by ARM, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently, see *c0, Main ID Register (MIDR)* on page AppxH-34. Processors implemented by ARM have an Implementer code of 0x41.

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the device.

ARMv7 requires all implementations to use the CPUID scheme, described in Chapter B5 *The CPUID Identification Scheme*, and an implementation is described by the MIDR and the CPUID registers.

Note

For an ARMv7 implementation by ARM, the MIDR is interpreted as:

Bits [31:24] Implementer code, must be 0x41.
Bits [23:20] Major revision number, rX.
Bits [19:16] Architecture code, must be 0xF.
Bits [15:4] ARM part number.
Bits [3:0] Minor revision number, pY.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アーキテクチャ、ビット [19:16]

このフィールドに許可される値を、表 B4- 11 に示します。

表 B4- 11 アーキテクチャコード

ビット [19:16]	アーキテクチャ
0x1	ARMv4
0x2	ARMv4T
0x3	ARMv5（廃止）
0x4	ARMv5T
0x5	ARMv5TE
0x6	ARMv5TEJ
0x7	ARMv6
0xF	CPUID 方式で定義

他のすべての値は ARM によって予約されており、使用できません。

主要部品番号、ビット [15:4]

デバイスの、実装定義の主要部品番号。

注

ARM によって実装されたプロセッサでは、主要部品番号の上位 4 ビットが 0x0 または 0x7 の場合は、バリアントとアーキテクチャについて異なるエンコードが使用されます。*c0、メイン ID レジスタ (MIDR)* P. 付録 H- 34 を参照して下さい。ARM によって実装されたプロセッサの実装者コードは 0x41 です。

リビジョン、ビット [3:0]

デバイスの、実装定義のリビジョン番号。

ARMv7 では、すべての実装において、第 B5 章*「CPUID 識別方式」*で説明されている CPUID 方式を使用する必要があります。実装は、MIDR レジスタと CPUID レジスタにより示されます。

注

ARM による ARMv7 実装の場合は、MIDR が次のように解釈されます。

ビット [31:24] 実装者コード、0x41 にする必要があります。
ビット [23:20] メジャーリビジョン番号、rX
ビット [19:16] アーキテクチャコード、0xF にする必要があります。
ビット [15:4] ARM 部品番号
ビット [3:0] マイナーリビジョン番号、pY

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the MIDR

To access the MIDR you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c0,c0,0    ; Read CP15 Main ID Register
```

B4.6.7 c0, Cache Type Register (CTR)

The Cache Type Register, CTR, provides information about the architecture of the caches.

The CTR is:

- a 32-bit read-only register
- accessible only in privileged modes.

The format of the CTR is changed from ARMv7. The ARMv7 format of the register is indicated by bits [31:29] being set to 0b100. For details of the format of the Cache Type Register in versions of the ARM architecture before ARMv7 see *c0, Cache Type Register (CTR)* on page AppxH-35.

In ARMv7, the format of the CTR is:

31	29	28	27	24	23	20	19	16	15										4	3	0
1	0	0	0	CWG		ERG		DminLine	1	0	0	0	0	0	0	0	0	0	0	IminLine	

Bits [31:29] Set to 0b100 for the ARMv7 register format. Set to 0b000 for the format used in ARMv6 and earlier.

Bit [28] RAZ.

CWG, bits [27:24]

Cache Writeback Granule. Log_2 of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

A value of 0b0000 indicates that the CTR does not provide Cache Writeback Granule information and either:

- the architectural maximum of 512 words (2Kbytes) must be assumed
- the Cache Writeback Granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

Values greater than 0b1001 are reserved.

ERG, bits [27:24]

Exclusives Reservation Granule. Log_2 of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions. For more information, see *Tagging and the size of the tagged memory block* on page A3-20.

A value of 0b0000 indicates that the CTR does not provide Exclusives Reservation Granule information and the architectural maximum of 512 words (2Kbytes) must be assumed.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

MIDR へのアクセス

MIDR にアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,0,<Rt>,c0,c0,0    ; Read CP15 Main ID Register
```

B4.6.7 c0、キャッシュタイプ レジスタ (CTR)

キャッシュタイプ レジスタ (CTR) は、キャッシュのアーキテクチャに関する情報を提供します。

CTR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能

CTR のフォーマットは ARMv7 で変更されました。このレジスタの ARMv7 フォーマットは、ビット [31:29] が 0b100 にセットされていることで示されます。ARMv7 以前のバージョンの ARM アーキテクチャにおけるキャッシュタイプ レジスタのフォーマットの詳細については、*c0、キャッシュタイプ レジスタ (CTR)* P. 付録 H-35 を参照して下さい。

ARMv7 における CTR のフォーマットは次のとおりです。

31	29	28	27	24	23	20	19	16	15										4	3	0
1	0	0	0	CWG		ERG		DminLine	1	0	0	0	0	0	0	0	0	0	0	IminLine	

ビット [31:29] ARMv7 レジスタフォーマットの場合は、0b100 にセットします。ARMv6 およびそれ以前で使用されるフォーマットの場合は、0b000 にセットします。

ビット [28] RAZ

CWG、ビット [27:24]

キャッシュライトバックの粒度。キャッシュエントリに含まれるメモリ位置が変更された場合、そのキャッシュエントリの追い出しにより上書きされる可能性があるメモリの最大サイズを、ワード単位で示した数値の Log_2 です。

0b0000 の値は、CTR でキャッシュライトバックの粒度情報が提供されず、次のいずれかであることを意味します。

- アーキテクチャ上の上限である、512 ワード (2KB) を前提とする必要がある。
- キャッシュライトバックの粒度は、キャッシュサイズ ID レジスタにエンコードされている最大キャッシュライン サイズから特定できる。

0b1001 より大きい値は予約されています。

ERG、ビット [23:20]

排他予約の粒度。排他ロード命令と排他ストア命令用に実装されている予約粒度の最大サイズを、ワード単位で示した数値の Log_2 です。詳細については、*タグ付けとタグ付けされたメモリブロックのサイズ* P. A3-20 を参照して下さい。

0b0000 の値は、CTR で排他予約の粒度情報が提供されず、アーキテクチャ上の上限である 512 ワード (2KB) を前提とする必要があることを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Values greater than 0b1001 are reserved.

DminLine, bits [19:16]

Log_2 of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the processor.

Bit [15] RAO.

Bits [14:4] RAZ.

IminLine, bits [3:0]

Log_2 of the number of words in the smallest cache line of all the instruction caches that are controlled by the processor.

Accessing the CTR

To access the CTR you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 1. For example

```
MRC p15,0,<Rt>,c0,c0,1    ; Read CP15 Cache Type Register
```

B4.6.8 c0, TCM Type Register (TCMTR)

The TCM Type Register, TCMTR, provides information about the implementation of the TCM.

The TCMTR is:

- a 32-bit read-only register
- accessible only in privileged modes.

From ARMv7:the

- TCMTR must be implemented
- when the ARMv7 format is used, the meaning of register bits [28:0] is IMPLEMENTATION DEFINED
- the ARMv6 format of the TCM Type Register remains a valid usage model
- if no TCMs are implemented the ARMv6 format must be used to indicate zero-sized TCMs.

The ARMv7 format of the TCMTR is:

31 29	28 0
1 0 0	IMPLEMENTATION DEFINED

Bits [31:29] Set to 0b100 for the ARMv7 register format. Set to 0b000 for the format used in ARMv6 and earlier.

Bits [28:0] IMPLEMENTATION DEFINED in the ARMv7 register format.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b1001 より大きい値は予約されています。

DminLine、ビット [19:16]

プロセッサで制御されるすべてのデータキャッシュと統一キャッシュの、最小キャッシュラインのワード数の Log_2 です。

ビット [15] RAO

ビット [14:4] RAZ

IminLine、ビット [3:0]

プロセッサで制御されるすべての命令キャッシュの、最小キャッシュラインのワード数の Log_2 です。

CTR へのアクセス

CTR にアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 1 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,0,<Rt>,c0,c0,1    ; Read CP15 Cache Type Register
```

B4.6.8 c0、TCM タイプレジスタ (TCMTR)

TCM タイプレジスタ (TCMTR) は、TCM の実装に関する情報を提供します。

TCMTR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能

ARMv7 以降では、次の規則が適用されます。

- TCMTR を実装する必要があります。
- ARMv7 フォーマットが使用されている場合、レジスタのビット [28:0] の意味は実装定義です。
- TCM タイプレジスタの ARMv6 フォーマットは、有効な使用モデルとして残されています。
- TCM が実装されていない場合は、ARMv6 フォーマットを使用してサイズ 0 の TCM を示す必要があります。

TCMTR の ARMv7 フォーマットは次のとおりです。

31 29	28 0
1 0 0	実装定義

ビット[31:29] ARMv7 レジスタフォーマットの場合は、0b100 にセットします。ARMv6 およびそれ以前で使用されるフォーマットの場合は、0b000 にセットします。

ビット[28:0] ARMv7 レジスタフォーマットでは実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If no TCMs are implemented, the TCMTR must be implemented with this ARMv6 format:

31 29	28 19	18 16	15 3	2 0
0 0 0	UNKNOWN	0 0 0	UNKNOWN	0 0 0

For details of the ARMv6 optional implementation of the TCM Type Register see *c0, TCM Type Register (TCMTR)* on page AppxG-33.

Accessing the TCMTR

To access the TCMTR you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 2. For example:

```
MRC p15,0,<Rt>,c0,c0,2    ; Read CP15 TCM Type Register
```

B4.6.9 c0, MPU Type Register (MPUIR)

The MPU Type Register, MPUIR, identifies the features of the MPU implementation. In particular it identifies:

- whether the MPU implements:
  - — a Unified address map, also referred to as a von Neumann architecture
  - — separate Instruction and Data address maps, also referred to as a Harvard architecture.
- the number of memory regions implemented by the MPU.

The MPUIR is:

- a 32-bit read-only register
- accessible only in privileged modes
- implemented only when the PMSA is implemented.

The format of the MPUIR is:

31 24	23 16	15 8	7 1	0
UNKNOWN	IRegion	DRegion	UNKNOWN	nU

Bits [31:24] UNKNOWN.

IRegion, bits [23:16]

Specifies the number of Instruction regions implemented by the MPU.

If the MPU implements a Unified memory map this field is UNK/SBZ.

DRegion, bits [15:8]

Specifies the number of Data or Unified regions implemented by the MPU.

If this field is zero, no MPU is implemented, and the default memory map is in use.

Bits [7:1] UNKNOWN.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

TCM が実装されていない場合は、この ARMv6 フォーマットを使用して TCMTR を実装する必要があります。

31 29	28 19	18 16	15 3	2 0
0 0 0	未知	0 0 0	未知	0 0 0

TCM タイプレジスタの ARMv6 オプション実装の詳細については、*c0、TCM タイプレジスタ (TCMTR)* P. 付録 G-33 を参照して下さい。

TCMTR へのアクセス

TCMTR にアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 2 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,0,<Rt>,c0,c0,2    ; Read CP15 TCM Type Register
```

B4.6.9 c0、MPU タイプレジスタ (MPUIR)

MPU タイプレジスタ (MPUIR) は、MPU 実装の機能を識別します。具体的には、次の情報が識別されます。

- MPU に次のものが実装されているかどうか。
  - – 統一アドレスマップ（フォン ノイマン アーキテクチャとも呼ばれます）
  - – 独立した命令アドレスマップとデータ アドレスマップ（ハーバードアーキテクチャとも呼ばれます）
- MPU により実装されているメモリ領域の数

MPUIR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- PMSA が実装されている場合にのみ実装されます。

MPUIR のフォーマットは次のとおりです。

31 24	23 16	15 8	7 1	0
未知	IRegion	DRegion	未知	nU

ビット [31:24] 未知

IRegion、ビット [23:16]

MPU によって実装されている命令領域の数を示します。

MPU で統一メモリマップが実装されている場合、このフィールドは UNK/SBZ です。

DRegion、ビット [15:8]

MPU によって実装されているデータ領域または統一領域の数を示します。

このフィールドが 0 の場合は、MPU が実装されておらず、デフォルトのメモリマップが使用されています。

ビット [7:1] 未知

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

nU, bit [0] Not Unified MPU. Indicates whether the MPU implements a unified memory map:

nU == 0 Unified memory map. Bits [23:16] of the register are zero.

nU == 1 Separate Instruction and Data memory maps.

Accessing the MPUIR

To access the MPUIR you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 4. For example:

```
MRC p15,0,<Rt>,c0,c0,4    ; Read CP15 MPU Type Register
```

B4.6.10 c0, Multiprocessor Affinity Register (MPIDR)

The Multiprocessor Affinity Register, MPIDR, provides an additional processor identification mechanism for scheduling purposes in a multiprocessor system. In a uniprocessor system ARM recommends that this register returns a value of 0.

The MPIDR is:

- a 32-bit read-only register
- accessible only in privileged modes
- introduced in ARMv7.

The format of the MPIDR is:

31–24	23–16	15–8	7–0
0 0 0 0 0 0 0 0	Affinity level 2	Affinity level 1	Affinity level 0

Note

In the MIDR bit definitions, a *processor in the system* can be a physical processor or a virtual processor.

Bits [31:24] Reserved, RAZ.

Affinity level 2, bits [23:16]

The least significant affinity level field, for this processor in the system.

Affinity level 1, bits [15:8]

The intermediate affinity level field, for this processor in the system.

Affinity level 0, bits [7:0]

The most significant level field, for this processor in the system.

In the system as a whole, for each of the affinity level fields, the assigned values must start at 0 and increase monotonically.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

nU、ビット [0]

非統一 MPU。MPU で統一メモリマップが実装されているかどうかを示します。

nU = 0 統一メモリマップ。レジスタのビット [23:16] は 0 です。

nU = 1 独立した命令メモリマップとデータメモリマップ。

MPUIR へのアクセス

MPUIR にアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 4 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,0,<Rt>,c0,c0,4    ; Read CP15 MPU Type Register
```

B4.6.10 c0、マルチプロセッサ類似性レジスタ (MPIDR)

マルチプロセッサ類似性レジスタ (MPIDR) は、マルチプロセッサシステム内のスケジューリング用の補助的なプロセッサ識別機構を提供します。ユニプロセッサシステムでは、このレジスタから 0 の値を返すことをお勧めします。

MPIDR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- ARMv7 で導入されました。

MPIDR のフォーマットは次のとおりです。

31–24	23–16	15–8	7–0
0 0 0 0 0 0 0 0	類似性レベル 2	類似性レベル 1	類似性レベル 0

注

MIDR ビット定義では、*システム内のプロセッサ*を物理プロセッサまたは仮想プロセッサにすることができます。

ビット[31:24] 予約、RAZ

類似性レベル 2、ビット [23:16]

最下位類似性レベルフィールド（システム内でこのプロセッサについて）。

類似性レベル 1、ビット [15:8]

中間類似性レベルフィールド（システム内でこのプロセッサについて）。

類似性レベル 0、ビット [7:0]

最上位類似性レベルフィールド（システム内でこのプロセッサについて）。

システム全体で、類似性レベルフィールドごとに、割り当てる値は 0 から始まって単調増加する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Increasing monotonically means that:

- There must not be any gaps in the sequence of numbers used.
- A higher value of the field includes any properties indicated by all lower values of the field.

When matching against an affinity level field, scheduler software checks for a value equal to or greater than a required value.

Recommended use of the MPIDR includes a description of an example multiprocessor system and the affinity level field values it might use.

The interpretation of these fields is IMPLEMENTATION DEFINED, and must be documented as part of the documentation of the multiprocessor system. ARM recommends that this register might be used as described in the next subsection.

The software mechanism to discover the total number of affinity numbers used at each level is IMPLEMENTATION DEFINED, and is part of the general system identification task.

Recommended use of the MPIDR

In a multiprocessor system the register might provide two important functions:

- Identifying special functionality of a particular processor in the system. In general, the actual meaning of the affinity level fields is not important. In a small number of situations, an affinity level field value might have a special IMPLEMENTATION DEFINED significance. Possible examples include booting from reset and power-down events.
- Providing affinity information for the scheduling software, to help the scheduler run an individual thread or process on either:
  - — the same processor, or as similar a processor as possible, as the processor it was running on previously
  - — a processor on which a related thread or process was run.

Note

A monotonically increasing single number ID mechanism provides a convenient index into software arrays and for accessing the interrupt controller. This might be:

- performed as part of the boot sequence
- stored as part of the local storage of threads.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

単調増加の意味は次のとおりです。

- 使用する数値の列に空白が存在しない。
- より高い値のフィールドには、より低い値のフィールドで示されるすべてのプロパティが含まれる。

類似性レベルフィールドに対する一致では、スケジューラソフトウェアによって必要な値以上かどうかがチェックされます。

マルチプロセッサシステムの例と、そのシステムで使用される可能性のある類似性レベルフィールドの値を、*MPIDR の推奨される使用法*に示します。

これらのフィールドの解釈は実装定義で、マルチプロセッサシステムの説明書に記載する必要があります。このレジスタは、次のサブセクションで説明されているように使用することをお勧めします。

各レベルで使用される類似性番号の総数を検出するためのソフトウェア機構は実装定義で、一般的なシステム識別タスクの一部です。

MPIDR の推奨される使用法

マルチプロセッサシステムでは、このレジスタによって次の 2 つの重要な機能が提供されます。

- システムの特定のプロセッサが持つ、特殊な機能の識別。一般的に、類似性レベルフィールドの実際の意味は重要ではありません。一部の状況では、類似性レベルフィールドの値が特殊な実装定義の意味を持つことがあります。この例としては、リセットやパワーダウン イベントからのブートが挙げられます。
- スケジューリングソフトウェアが、個別のスレッドまたはプロセスを次のようなプロセッサで実行を試みるときに役立てるため、スケジューラに類似性情報を提供します。
  - – 以前にスレッドまたはプロセスを実行していたプロセッサと同じ、または可能な限り類似のプロセッサ
  - – 関連するスレッドまたはプロセスが実行されていたプロセッサ

注

単調に増加する単一番号 ID 機構によって、ソフトウェアアレイと、割り込みコントローラへのアクセスにとって便利なインデクスが提供されます。この操作は、次の方法で実行できます。

- ブートシーケンスの一部として実行される。
- スレッドのローカル記憶域の一部としてストアされる。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

MPIDR provides a mechanism with up to three levels of affinity information, but the meaning of those levels of affinity is entirely IMPLEMENTATION DEFINED. The levels of affinity provided can have different meanings. Table B4-12 shows two possible implementations:

Table B4-12 Possible implementations of the affinity levels

Affinity Level	Example system 1	Example system 2
0	Virtual CPUs in a in a multi-threaded processor	Processors in an SMP cluster
1	Processors in an *Symmetric Multi Processor* (SMP) cluster	Clusters with a system
2	Clusters in a system	No meaning, fixed as 0.

The scheduler maintains affinity level information for all threads and processes. When it has to reschedule a thread or process the scheduler:

- looks for an available processor that matches at all three affinity levels
- if this fails, it might look for a processor that matches at levels 2 and 3 only
- if it still cannot find an available processor it might look for a match at level 3 only.

A multiprocessor system corresponding to Example system 1 in Table B4-12 might implement affinity values as shown in Table B4-13:

Table B4-13 Example of possible affinity values at different affinity levels

Affinity level 2, Cluster level	Affinity level 1, Processor level	Affinity level 0, Virtual CPU level
0	0	0, 1
0	1	0, 1
0	2	0, 1
0	3	0, 1
1	0	0, 1
1	1	0, 1
1	2	0, 1
1	3	0, 1

Accessing the MPIDR

To access the MPIDR you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 5. For example:

```
MRC p15,0,<Rt>,c0,c0,5    ; Read Multiprocessor Affinity Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

MPIDR によって、最大で 3 レベルの類似性情報を持つ機構が提供されますが、これらのレベルの類似性の意味は、完全に実装定義です。提供する類似性のレベルによって意味を変えることができます。実現可能な 2 つの実装を、表 B4-12 に示します。

表 B4-12 実現可能な類似性レベルの実装

類似性レベル	サンプルシステム 1	サンプルシステム 2
0	マルチスレッド プロセッサ内の仮想 CPU	SMP クラスタ内のプロセッサ
1	*対称型マルチプロセッサ*(SMP) クラスタ内のプロセッサ	システムを伴うクラスタ
2	システム内のクラスタ	意味なし、常に 0

スケジューラは、すべてのスレッドとプロセスに関する類似性レベル情報を保持します。スレッドまたはプロセスを再スケジュールする必要がある場合は、スケジューラが次の処理を実行します。

- 3 つすべての類似性レベルに一致する使用可能なプロセッサを探す。
- これに失敗した場合は、レベル 2 と 3 でのみ一致するプロセッサを探す。
- それでも使用可能なプロセッサが見つからない場合は、レベル 3 でのみ一致するプロセッサを探す。

表 B4-12 のサンプルシステム 1 に対応するマルチプロセッサシステムでは、表 B4-13 に示すような類似性の値を実装することができます。

表 B4-13 複数の類似性レベルで使用可能な類似性の値の例

類似性レベル 2、クラスタレベル	類似性レベル 1、プロセッサレベル	類似性レベル 0、仮想 CPU レベル
0	0	0, 1
0	1	0, 1
0	2	0, 1
0	3	0, 1
1	0	0, 1
1	1	0, 1
1	2	0, 1
1	3	0, 1

MPIDR へのアクセス

MPIDR にアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 5 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,0,<Rt>,c0,c0,5    ; Read Multiprocessor Affinity Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.11 c0, Cache Size ID Registers (CCSIDR)

The Cache Size ID Registers, CCSIDR, provide information about the architecture of the caches.

The CCSIDR registers are:

- 32-bit read-only registers
- accessible only in privileged modes
- introduced in ARMv7.

One CCSIDR is implemented for each cache that can be accessed by the processor. CSSELR selects which Cache Size ID register is accessible, see *c0, Cache Size Selection Register (CSSELR)* on page B4-43.

The format of a CCSIDR is:

31	30	29	28	27 … 13	12 … 3	2 … 0
WT	WB	RA	WA	NumSets	Associativity	LineSize

WT, bit [31] Indicates whether the cache level supports Write-Through, see Table B4-14.

WB, bit [30] Indicates whether the cache level supports Write-Back, see Table B4-14.

RA, bit [29] Indicates whether the cache level supports Read-Allocation, see Table B4-14.

WA, bit [28] Indicates whether the cache level supports Write-Allocation, see Table B4-14.

Table B4-14 WT, WB, RA and WA bit values

WT, WB, RA or WA bit value	Meaning
0	Feature not supported
1	Feature supported

NumSets, bits [27:13]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Associativity, bits [12:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B4.6.11 c0、キャッシュサイズ ID レジスタ (CCSIDR)

キャッシュサイズ ID レジスタ (CCSIDR) は、キャッシュのアーキテクチャに関する情報を提供します。

CCSIDR レジスタの特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- ARMv7 で導入されました。

プロセッサからアクセス可能なキャッシュごとに、1 つの CCSIDR が実装されます。CSSELR によって、アクセス可能なキャッシュサイズ ID レジスタが選択されます。*c0、キャッシュサイズ選択レジスタ (CSSELR)* P. B4- 43 を参照して下さい。

CCSIDR のフォーマットは次のとおりです。

31	30	29	28	27 … 13	12 … 3	2 … 0
WT	WB	RA	WA	NumSets	Associativity	LineSize

WT、ビット [31] そのキャッシュレベルでライトスルーがサポートされているかどうかを示します。表 B4- 14 を参照して下さい。

WB、ビット [30] そのキャッシュレベルでライトバックがサポートされているかどうかを示します。表 B4- 14 を参照して下さい。

RA、ビット [29] そのキャッシュレベルで読み出し割り当てがサポートされているかどうかを示します。表 B4- 14 を参照して下さい。

WA、ビット [28] そのキャッシュレベルで書き込み割り当てがサポートされているかどうかを示します。表 B4- 14 を参照して下さい。

表 B4- 14 WT、WB、RA、WA ビットの値

WT、WB、RA、WA ビットの値	意味
0	機能がサポートされていない
1	機能がサポートされている

NumSets、ビット [27:13]

（キャッシュのセット数）－ 1。したがって、0 の値はキャッシュが 1 セットであることを意味します。セット数を 2 の累乗にする必要はありません。

Associativity、ビット [12:3]

（キャッシュのアソシエティビティ）－ 1。したがって、0 の値はアソシエティビティが 1 であることを意味します。アソシエティビティを 2 の累乗にする必要はありません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

LineSize, bits [2:0]

(Log_2(Number of words in cache line)) -2. For example:

- For a line length of 4 words: $Log_2(4) = 2$, LineSize entry = 0.
  This is the minimum line length.
- For a line length of 8 words: $Log_2(8) = 3$, LineSize entry = 1.

Accessing the currently selected CCSIDR

The CSSELR selects a CCSIDR, see *c0, Cache Size Selection Register (CSSELR)* on page B4-43. To access the currently-selected CCSIDR you read the CP15 registers with `<opc1>` set to 1, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,1,<Rt>,c0,c0,0    ; Read current CP15 Cache Size ID Register
```

Accessing the CCSIDR when the value in CSSELR corresponds to a cache that is not implemented returns an UNKNOWN value.

B4.6.12 c0, Cache Level ID Register (CLIDR)

The Cache Level ID Register, CLIDR:

- identifies the type of cache, or caches, implemented at each level, up to a maximum of eight levels
- identifies the Level of Coherency and Level of Unification for the cache hierarchy.

The CLIDR is:

- a 32-bit read-only register
- accessible only in privileged modes
- introduced in ARMv7.

The format of the CLIDR is:

31	30	29 27	26 24	23 21	20 18	17 15	14 12	11 9	8 6	5 3	2 0
0	0	LoUU	LoC	LoUIS	Ctype7	Ctype6	Ctype5	Ctype4	Ctype3	Ctype2	Ctype1

Bits [31:30] RAZ.

LoUU, bits [29:27]

Level of Unification Uniprocessor for the cache hierarchy, see *Clean, Invalidate, and Clean and Invalidate* on page B2-11.

LoC, bits [26:24]

Level of Coherency for the cache hierarchy, see *Clean, Invalidate, and Clean and Invalidate* on page B2-11.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

LineSize、ビット [2:0]

(Log_2(キャッシュラインのワード数)) – 2。次に例を示します。

- ライン長が 4 ワードの場合：$Log_2(4) = 2$、LineSize エントリ = 0
  これは最小ライン長です。
- ライン長が 8 ワードの場合：$Log_2(8) = 3$、LineSize エントリ = 1

現在選択されている CCSIDR へのアクセス

CSSELR によって CCSIDR が選択されます。*c0、キャッシュサイズ選択レジスタ (CSSELR)* P. B4- 43 を参照して下さい。現在選択されている CCSIDR にアクセスするには、`<opc1>` を 1、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,1,<Rt>,c0,c0,0    ; Read current CP15 Cache Size ID Register
```

CSSELR の値が、実装されていないキャッシュに対応している場合に、CCSIDR にアクセスすると、未知の値が返されます。

B4.6.12 c0、キャッシュレベル ID レジスタ (CLIDR)

キャッシュレベル ID レジスタ (CLIDR) の機能は次のとおりです。

- 最大 8 レベルのキャッシュについて、レベルごとに実装されているキャッシュタイプを識別する。
- キャッシュ階層のコヒーレンシレベルと統一レベルを識別する。

CLIDR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- ARMv7 で導入されました。

CLIDR のフォーマットは次のとおりです。

31	30	29 27	26 24	23 21	20 18	17 15	14 12	11 9	8 6	5 3	2 0
0	0	LoUU	LoC	LoUIS	Ctype7	Ctype6	Ctype5	Ctype4	Ctype3	Ctype2	Ctype1

ビット [31:30] RAZ

LoUU、ビット [29:27]

キャッシュ階層のユニプロセッサ統一レベル。*クリーニング、無効化、およびクリーニングと無効化* P. B2- 11 を参照して下さい。

LoC、ビット [26:24]

キャッシュ階層のコヒーレンシレベル。*クリーニング、無効化、およびクリーニングと無効化* P. B2- 11 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

LoUIS, bits [23:21]

Level of Unification Inner Shareable for the cache hierarchy, see *Clean, Invalidate, and Clean and Invalidate* on page B2-11. This field is RAZ in implementations that do not implement the Multiprocessing extension.

CtypeX, bits [3(x - 1) + 2:3(x - 1)], for x = 1 to 7

Cache type fields. Indicate the type of cache implemented at each level, from Level 1 up to a maximum of seven levels of cache hierarchy. The Level 1 cache type field, Ctype1, is bits [2:0], see register diagram. Table B4-15 shows the possible values for each CtypeX field.

Table B4-15 Ctype bit values

CtypeX bits	Meaning, cache implemented at this level
000	No cache
001	Instruction cache only
010	Data cache only
011	Separate instruction and data caches
100	Unified cache
101, 11X	Reserved

If you read the Cache type fields from Ctype1 upwards, once you have seen a value of 0b000, no caches exist at further out levels of the hierarchy. So, for example, if Ctype3 is the first Cache type field with a value of 0b000, the values of Ctype4 to Ctype7 must be ignored.

The CLIDR describes only the caches that are under the control of the processor.

Accessing the CLIDR

To access the CLIDR you read the CP15 registers with `<opc1>` set to 1, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 1. For example:

```
MRC p15,1,<Rt>,c0,c0,1    ; Read CP15 Cache Level ID Register
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

LoUIS、ビット [23:21]

キャッシュ階層の内部共有可能統一レベル。*クリーニング、無効化、およびクリーニングと無効化* P. B2-11 を参照して下さい。このフィールドは、マルチプロセッシング拡張機能を備えていない実装では RAZ です。

CtypeX、ビット [3(x − 1) + 2:3(x − 1)]、x = 1 〜 7 の場合

キャッシュタイプ フィールド。キャッシュ階層のレベル 1 から、最大でレベル 7 までのキャッシュレベルごとに、実装されるキャッシュタイプを示します。レベル 1 キャッシュタイプ フィールドの Ctype1 は、ビット [2:0] です。レジスタ図を参照して下さい。各 CtypeX フィールドの可能な値を、表 B4- 15 に示します。

表 B4- 15 Ctype ビットの値

CtypeX ビット	意味、このレベルに実装されているキャッシュ
000	キャッシュなし
001	命令キャッシュのみ
010	データキャッシュのみ
011	独立した命令キャッシュとデータキャッシュ
100	統一キャッシュ
101、11X	予約

Ctype1 から上に向かってキャッシュタイプ フィールドを読み出す場合は、0b000 の値を読み出した時点で、階層のその上のレベルにキャッシュが存在しないことを意味します。したがって、例えば、Ctype3 が 0b000 の値を持つ最初のキャッシュタイプ フィールドであれば、Ctype4 〜 Ctype7 の値を無視する必要があります。

CLIDR では、プロセッサの制御下にあるキャッシュのみが示されます。

CLIDR へのアクセス

CLIDR にアクセスするには、`<opc1>` を 1、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 1 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,1,<Rt>,c0,c0,1    ; Read CP15 Cache Level ID Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.13 c0, IMPLEMENTATION DEFINED Auxiliary ID Register (AIDR)

The IMPLEMENTATION DEFINED Auxiliary ID Register, AIDR, provides implementation-specific ID information. The value of this register must be used in conjunction with the value of the MIDR.

The IMPLEMENTATION DEFINED AIDR is:

- a 32-bit read-only register
- accessible only in privileged modes
- introduced in ARMv7.

The format of the AIDR is IMPLEMENTATION DEFINED.

Accessing the AIDR

To access the AIDR you read the CP15 registers with `<opc1>` set to 1, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 7. For example:

```
MRC p15,1,<Rt>,c0,c0,7    ; Read IMPLEMENTATION DEFINED Auxiliary ID Register
```

B4.6.14 c0, Cache Size Selection Register (CSSELR)

The Cache Size Selection Register, CSSELR, selects the current CCSIDR. An ARMv7 implementation must include a CCSIDR for every implemented cache that is under the control of the processor. The CSSELR identifies which CCSIDR can be accessed, by specifying, for the required cache:

- the cache level
- the cache type, either:
  - — instruction cache.
  - — Data cache. The data cache argument is also used for a unified cache.

CSSELR is:

- a 32-bit read/write register
- accessible only in privileged modes
- introduced in ARMv7.

The format of the CSSELR is:

Bits [31:4] UNK/SBZP.

Level, bits [3:1]

Cache level of required cache. Permitted values are from 0b000, indicating Level 1 cache, to 0b110 indicating Level 7 cache.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.13 c0、実装定義の補助 ID レジスタ (AIDR)

実装定義の補助 ID レジスタ (AIDR) は、実装固有の ID 情報を提供します。このレジスタの値は、MIDR の値と組み合わせて使用する必要があります。

実装定義の AIDR の特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- ARMv7 で導入されました。

AIDR のフォーマットは実装定義です。

AIDR へのアクセス

AIDR にアクセスするには、`<opc1>` を 1、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 7 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15,1,<Rt>,c0,c0,7    ; Read IMPLEMENTATION DEFINED Auxiliary ID Register
```

B4.6.14 c0、キャッシュサイズ選択レジスタ (CSSELR)

キャッシュサイズ選択レジスタ (CSSELR) は、現在の CCSIDR を選択します。ARMv7 実装には、プロセッサの制御下にある、実装されているすべてのキャッシュについて CCSIDR を含める必要があります。CSSELR では、必要なキャッシュについて次の要素を指定することによって、アクセス可能な CCSIDR が識別されます。

- キャッシュレベル
- 次のいずれかのキャッシュタイプ
  - – 命令キャッシュ
  - – データキャッシュ。データキャッシュの引数は、統一キャッシュにも使用されます。

CSSELR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- ARMv7 で導入されました。

CSSELR のフォーマットは次のとおりです。

ビット [31:4] UNK/SBZP

レベル、ビット [3:1]

必要なキャッシュのキャッシュレベル。許される値は、レベル 1 キャッシュを示す 0b000 から、レベル 7 キャッシュを示す 0b110 までです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

InD, bit [0]

Instruction not data bit. Permitted values are:

0 Data or unified cache

1 Instruction cache.

If CSSELR is set to indicate a cache that is not implemented, the result of reading the current CCSIDR is UNPREDICTABLE.

Accessing CSSELR

To access CSSELR you read or write the CP15 registers with `<opc1>` set to 2, `<CRn>` set to c0, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,2,<Rt>,c0,c0,0    ; Read Cache Size Selection Register
MCR p15,2,<Rt>,c0,c0,0    ; Write Cache Size Selection Register
```

B4.6.15 CP15 c1, System control registers

The CP15 c1 registers are used for system control. Figure B4-5 shows the CP15 c1 registers.

Figure B4-5 CP15 c1 registers in a PMSA implementation

All CP15 c1 register encodings not shown in Figure B4-5 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B4-27.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

InD、ビット [0]

命令 / データビット。許される値は次のとおりです。

0 データまたは統一キャッシュ

1 命令キャッシュ

CSSELR が、実装されていないキャッシュを示すように設定されている場合、現在の CCSIDR の読み出し結果は予測不能です。

CSSELR へのアクセス

CSSELR にアクセスするには、`<opc1>` を 2、`<CRn>` を c0、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,2,<Rt>,c0,c0,0    ; Read Cache Size Selection Register
MCR p15,2,<Rt>,c0,c0,0    ; Write Cache Size Selection Register
```

B4.6.15 CP15 c1、システム制御レジスタ

CP15 c1 レジスタは、システム制御に使用されます。CP15 c1 レジスタを、図 B4-5 に示します。

図 B4-5 PMSA 実装の CP15 c1 レジスタ

図 B4-5 に示されていない CP15 c1 レジスタのすべてのエンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.16 c1, System Control Register (SCTLR)

The System Control Register, SCTLR, provides the top level control of the system, including its memory system.

The SCTLR:

- Is a 32-bit read/write register, with different access rights for some bits of the register.

  In ARMv7, some bits in the register are read-only. These bits relate to non-configurable features of an ARMv7 implementation, and are provided for compatibility with previous versions of the architecture.

- Is accessible only in privileged modes.

- Has a defined reset value. The reset value is IMPLEMENTATION DEFINED, see *Reset value of the SCTLR* on page B4-49.

Control bits in the SCTLR that are not applicable to a PMSA implementation read as the value that most closely reflects that implementation, and ignore writes.

In an ARMv7-R implementation the format of the SCTLR is:

IE, bit [31] Instruction Endianness. This bit indicates the endianness of the instructions issued to the processor:

0 Little-endian byte ordering in the instructions

1 Big-endian byte ordering in the instructions.

When set, this bit causes the byte order of instructions to be reversed at runtime.

This bit is read-only. It is IMPLEMENTATION DEFINED which instruction endianness is used by an ARMv7-R implementation, and this bit must indicate the implemented endianness.

If IE == 1 and EE == 0, behavior is UNPREDICTABLE.

TE, bit [30] Thumb Exception enable. This bit controls whether exceptions are taken in ARM or Thumb state:

0 Exceptions, including reset, handled in ARM state

1 Exceptions, including reset, handled in Thumb state.

An implementation can include a configuration input signal that determines the reset value of the TE bit. If the implementation does not include a configuration signal for this purpose then this bit resets to zero in an ARMv7-R implementation.

For more information about the use of this bit see *Instruction set state on exception entry* on page B1-35.

Bits [29:28] RAZ/SBZP.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.16 c1、システム制御レジスタ (SCTLR)

システム制御レジスタ (SCTLR) は、メモリシステムを含むシステムのトップレベルの制御を提供します。

SCTLR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタで、一部のビットはアクセス権限が異なっています。

  ARMv7 では、このレジスタの一部のビットが読み出し専用です。これらのビットは、ARMv7 実装の構成不可機能に関係しており、以前のバージョンのアーキテクチャとの互換性を維持するために提供されています。

- 特権モードでのみアクセス可能

- リセット時の値が定義されています。リセット時の値は実装定義です（*SCTLR のリセット時の値* P. B4-49 を参照）。

PMSA 実装に適用されない SCTLR の制御ビットは、その実装を最も厳密に反映している値として読み出されます。書き込みは無視されます。

ARMv7-R 実装における SCTLR のフォーマットは次のとおりです。

IE、ビット[31] 命令のエンディアン形式。このビットは、プロセッサに発行された命令のエンディアン形式を示します。

0 命令のバイト順序はリトルエンディアンです。

1 命令のバイト順序はビッグエンディアンです。

このビットがセットされると、命令のバイト順序が実行時に反転されます。

このビットは読み出し専用です。ARMv7-R 実装でどの命令エンディアン形式が使用されるかは実装定義で、このビットは実装されているエンディアン形式を示す必要があります。

IE == 1 かつ EE == 0 の場合、動作は予測不能です。

TE、ビット[30] Thumb 例外イネーブル。このビットによって、ARM 状態と Thumb 状態のどちらで例外が取得されるかが制御されます。

0 例外は、リセットも含めて、ARM 状態で処理されます。

1 例外は、リセットも含めて、Thumb 状態で処理されます。

実装には、TE ビットのリセット時の値を決定する構成入力信号を含めることができます。ARMv7-R 実装では、この目的の構成信号が実装に含まれていなければ、このビットが 0 にリセットされます。

このビットの使用方法の詳細については、*例外の開始時点の命令セット状態* P. B1-35 を参照して下さい。

ビット[29:28] RAZ/SBZP

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

NMFI, bit [27]

Non-Maskable Fast Interrupts enable:

0 Fast interrupts (FIQs) can be masked in the CPSR

1 Fast interrupts are non-maskable.

This bit is read-only. It is IMPLEMENTATION DEFINED whether an implementation supports *Non-Maskable Fast Interrupts* (NMFIs):

- If NMFIs are not supported then this bit is RAZ/WI.
- If NMFIs are supported then this bit is determined a configuration input signal.

For more information, see *Non-maskable fast interrupts* on page B1-18.

Bit [26] RAZ/SBZP.

EE, bit [25] Exception Endianness bit. The value of this bit defines the value of the CPSR.E bit on entry to an exception vector, including reset. The permitted values of this bit are:

0 Little endian

1 Big endian.

This is a read/write bit. An implementation can include a configuration input signal that determines the reset value of the EE bit. If the implementation does not include a configuration signal for this purpose then this bit resets to zero.

If IE == 1 and EE == 0, behavior is UNPREDICTABLE.

VE, bit [24] Interrupt Vectors Enable bit. This bit controls the vectors used for the FIQ and IRQ interrupts. The permitted values of this bit are:

0 Use the FIQ and IRQ vectors from the vector table, see the V bit entry

1 Use the IMPLEMENTATION DEFINED values for the FIQ and IRQ vectors.

For more information, see *Vectored interrupt support* on page B1-32.

If the implementation does not support IMPLEMENTATION DEFINED FIQ and IRQ vectors then this bit is RAZ/WI.

Bit [23] RAO/SBOP.

U, bit [22] In ARMv7 this bit is RAO/SBOP, indicating use of the alignment model described in *Alignment support* on page A3-4.

For details of this bit in earlier versions of the architecture see *Alignment* on page AppxG-6.

FI, bit [21] Fast Interrupts configuration enable bit. This bit can be used to reduce interrupt latency in an implementation by disabling IMPLEMENTATION DEFINED performance features. The permitted values of this bit are:

0 All performance features enabled.

1 Low interrupt latency configuration. Some performance features disabled.

If the implementation does not support a mechanism for selecting a low interrupt latency configuration this bit is RAZ/WI.

For more information, see *Low interrupt latency configuration* on page B1-43.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

NMFI、ビット [27]

マスク不能高速割り込みイネーブル

0 CPSR で高速割り込み (FIQ) をマスクできます。

1 高速割り込みはマスク不可です。

このビットは読み出し専用です。実装で*マスク不能高速割り込み* (NMFI) がサポートされるかどうかは、実装定義です。

- NMFI がサポートされていない場合、このビットは RAZ/WI です。
- NMFI がサポートされている場合は、このビットが構成入力信号によって決定されます。

詳細については、*マスク不能高速割り込み* P. B1-18 を参照して下さい。

ビット [26] RAZ/SBZP

EE、ビット[25] 例外エンディアン形式ビット。このビットの値によって、リセットを含む例外ベクタの開始時の CPSR.E ビットの値が定義されます。このビットに許される値は次のとおりです。

0 リトルエンディアン

1 ビッグエンディアン

読み出し / 書き込みビットです。実装には、EE ビットのリセット時の値を決定する構成入力信号を含めることができます。この目的の構成信号が実装に含まれていなければ、このビットが 0 にリセットされます。

IE == 1 かつ EE == 0 の場合、動作は予測不能です。

VE、ビット[24] 割り込みベクタイネーブルビット。このビットによって、FIQ 割り込みと IRQ 割り込みで使用されるベクタが制御されます。このビットに許される値は次のとおりです。

0 ベクタテーブルから FIQ ベクタと IRQ ベクタを使用します（V ビットのエントリを参照）。

1 FIQ ベクタと IRQ ベクタの実装定義の値を使用します。

詳細については、*ベクタ割り込みのサポート* P. B1-32 を参照して下さい。

実装で実装定義の FIQ ベクタと IRQ ベクタがサポートされていない場合、このビットは RAZ/WI です。

ビット [23] RAO/SBOP

U、ビット[22] ARMv7 では、このビットは RAO/SBOP で、*アライメントのサポート* P. A3-4 で説明されているアライメントモデルが使用されることを示します。

以前のバージョンのアーキテクチャにおけるこのビットの詳細については、*アライメント* P. 付録 G-6 を参照して下さい。

FI、ビット[21] 高速割り込み構成イネーブルビット。このビットは、実装定義のパフォーマンス機能を非稼動にすることによって、実装内の割り込みレイテンシを低減するために使用できます。このビットに許される値は次のとおりです。

0 すべてのパフォーマンス機能が稼動しています。

1 低割り込みレイテンシ構成。一部のパフォーマンス機能が非稼動です。

実装で低割り込みレイテンシ構成の選択機構がサポートされていない場合、このビットは RAZ/WI です。

詳細については、*低割り込みレイテンシ構成* P. B1-43 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Bit [20] RAZ/SBZP.

DZ, bit [19] Divide by Zero fault enable bit. Any ARMv7-R implementation includes instructions to perform unsigned and signed division, see *SDIV* on page A8-310 and *UDIV* on page A8-468. This bit controls whether an integer divide by zero causes an Undefined Instruction exception:

0 Divide by zero returns the result zero, and no exception is taken

1 Attempting a divide by zero causes an Undefined Instruction exception on the `SDIV` or `UDIV` instruction.

Bit [18] RAO/SBOP.

BR, bit [17] Background Region bit. When the MPU is enabled this bit controls how an access that does not map to any MPU memory region is handled:

0 Any access to an address that is not mapped to an MPU region generates a Background Fault memory abort. This is the PMSAv6 behavior.

1 The default memory map is used as a background region:

- A privileged access to an address that does not map to an MPU region takes the properties defined for that address in the default memory map.
- An unprivileged access to an address that does not map to an MPU region generates a Background Fault memory abort.

For more information, see *Using the default memory map as a background region* on page B4-5.

Bit [16] RAO/SBOP.

Bit [15] RAZ/SBZP.

RR, bit [14] Round Robin bit. If the cache implementation supports the use of an alternative replacement strategy that has a more easily predictable worst-case performance, this bit selects it:

0 Normal replacement strategy, for example, random replacement

1 Predictable strategy, for example, round-robin replacement.

The RR bit must reset to 0.

The replacement strategy associated with each value of the RR bit is IMPLEMENTATION DEFINED.

If the implementation does not support multiple IMPLEMENTATION DEFINED replacement strategies this bit is RAZ/WI.

V, bit [13] Vectors bit. This bit selects the base address of the exception vectors:

0 Normal exception vectors, base address `0x00000000`.

1 High exception vectors (Hivecs), base address `0xFFFF0000`.

For more information, see *Exception vectors and the exception base address* on page B1-30.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ビット [20] RAZ/SBZP

DZ、ビット[19] 0 による除算フォールトイネーブル ビット。ARMv7-R 実装には、符号なしと符号付きの除算を実行するための命令が含まれています。*SDIV* P. A8-310 と *UDIV* P. A8-468 を参照して下さい。このビットによって、0 による整数除算が未定義命令例外を引き起こすかどうかが制御されます。

0 0 による除算では、結果として 0 が返され、例外は取得されません。

1 `SDIV` 命令または `UDIV` 命令で 0 による除算を試みると、未定義命令例外が発生します。

ビット [18] RAO/SBOP

BR、ビット[17] バックグラウンド領域ビット。MPU が許可されている場合は、このビットによって、どの MPU メモリ領域にもマッピングされていないアクセスがどのように処理されるかが制御されます。

0 MPU 領域にマップされていないアドレスへのアクセスによって、バックグラウンドフォールト メモリアボートが生成されます。これは、PMSAv6 の動作です。

1 デフォルトのメモリマップがバックグラウンド領域として使用されます。

- MPU 領域にマッピングされていないアドレスへの特権アクセスでは、デフォルトのメモリマップでそのアドレスについて定義されたプロパティが使用されます。
- MPU 領域にマップされていないアドレスへの非特権アクセスでは、バックグラウンドフォールト メモリアボートが生成されます。

詳細については、*デフォルトのメモリマップをバックグラウンド領域として使用する* P. B4-5 を参照して下さい。

ビット [16] RAO/SBOP

ビット [15] RAZ/SBZP

RR、ビット[14] ラウンドロビン ビット。キャッシュ実装で、ワーストケースのパフォーマンスの予測を容易にする代替置換方式の使用がサポートされている場合は、このビットによって次の手法が選択されます。

0 ランダム置換などの通常の置換方式

1 ラウンドロビン置換などの予測可能な置換方式

RR ビットは、0 にリセットする必要があります。

RR ビットのそれぞれの値に関連付けられている置換方式は、実装定義です。

実装で複数の実装定義の置換方式がサポートされていない場合、このビットは RAZ/WI です。

V、ビット[13] ベクタビット。このビットにより、次のように例外ベクタのベースアドレスが選択されます。

0 通常の例外ベクタ、ベースアドレス `0x00000000`。

1 上位例外ベクタ (Hivecs)、ベースアドレス `0xFFFF0000`

詳細については、*例外ベクタと例外ベースアドレス* P. B1-30 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

Use of the Hivecs setting, V == 1, is deprecated in an ARMv7-R implementation.

An implementation can include a configuration input signal that determines the reset value of the V bit. If the implementation does not include a configuration signal for this purpose then this bit resets to zero.

I, bit [12] Instruction cache enable bit: This is a global enable bit for instruction caches:

0 Instruction caches disabled

1 Instruction caches enabled.

If the system does not implement any instruction caches that can be accessed by the processor, at any level of the memory hierarchy, this bit is RAZ/WI.

If the system implements any instruction caches that can be accessed by the processor then it must be possible to disable them by setting this bit to 0.

Cache enabling and disabling on page B2-8 describes the effect of enabling the caches.

Z, bit [11] Branch prediction enable bit. This bit is used to enable branch prediction, also called program flow prediction:

0 Program flow prediction disabled

1 Program flow prediction enabled.

If program flow prediction cannot be disabled, this bit is RAO/WI.

If the implementation does not support program flow prediction then this bit is RAZ/WI.

SW, bit[10] SWP/SWPB enable bit. This bit enables the use of `SWP` and `SWPB` instructions:

0 `SWP` and `SWPB` are UNDEFINED

1 `SWP` and `SWPB` perform as described in section *SWP, SWPB* on page A8-432.

This bit is added as part of the Multiprocessing Extensions.

Note

At reset, this bit disables `SWP` and `SWPB`. This means that operating systems have to choose to use `SWP` and `SWPB`.

Bits [9:8] RAZ/SBZP.

B, bit [7] In ARMv7 this bit is RAZ/SBZP, indicating use of the endianness model described in *Endian support* on page A3-7.

For details of this bit in earlier versions of the architecture see *Endian support* on page AppxG-7 and *Endian support* on page AppxH-7.

Bits [6:3] RAO/SBOP.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

ARMv7-R 実装では、Hivecs 設定の V == 1 の使用は非推奨です。

実装には、V ビットのリセット時の値を決定する構成入力信号を含めることができます。この目的の構成信号が実装に含まれていなければ、このビットが 0 にリセットされます。

I、ビット[12] 命令キャッシュイネーブル ビット。命令キャッシュのグローバルイネーブル ビットです。

0 命令キャッシュが非稼動です。

1 命令キャッシュが稼動しています。

システムのメモリ階層のどのレベルにも、プロセッサからアクセス可能な命令キャッシュが実装されていない場合、このビットは RAZ/WI です。

システムにプロセッサからアクセス可能な命令キャッシュが実装されている場合は、このビットを 0 にセットしてそれらのキャッシュを非稼動にできる必要があります。

キャッシュの稼動の影響については、*キャッシュの稼動と非稼動* P. B2-8 で説明します。

Z、ビット[11] 分岐予測イネーブルビット。このビットは分岐予測（プログラムフロー予測とも呼ばれます）を可能にするために使用されます。

0 プログラムフロー予測を不可能にします。

1 プログラムフロー予測を可能にします。

プログラムフロー予測を不可能にできない場合、このビットは RAO/WI です。

実装でプログラムフロー予測がサポートされていない場合、このビットは RAZ/WI です。

SW、ビット[10] SWP/SWPB イネーブルビット。このビットは、`SWP` 命令と `SWPB` 命令の使用を可能にします。

0 `SWP` と `SWPB` は未定義です。

1 `SWP` と `SWPB` が、*SWP*、*SWPB* P. A8-432 で説明されているように動作します。

このビットは、マルチプロセッシング拡張機能の一部として追加されます。

注

リセット時には、このビットによって `SWP` と `SWPB` が不可能になります。これは、オペレーティングシステムで `SWP` および `SWPB` の使用を選択する必要があることを意味します。

ビット [9:8] RAZ/SBZP

B、ビット[7] ARMv7 では、このビットは RAZ/SBZP で、*エンディアンのサポート* P. A3-7 で説明されているエンディアン形式モデルの使用を示します。

以前のバージョンのアーキテクチャにおけるこのビットの詳細については、*エンディアンのサポート* P. 付録 G-7 と *エンディアンのサポート* P. 付録 H-7 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C, bit [2] Cache enable bit: This is a global enable bit for data and unified caches:

0 Data and unified caches disabled

1 Data and unified caches enabled.

If the system does not implement any data or unified caches that can be accessed by the processor, at any level of the memory hierarchy, this bit is RAZ/WI.

If the system implements any data or unified caches that can be accessed by the processor then it must be possible to disable them by setting this bit to 0.

Cache enabling and disabling on page B2-8 describes the effect of enabling the caches.

A, bit [1] Alignment bit. This is the enable bit for Alignment fault checking:

0 Alignment fault checking disabled

1 Alignment fault checking enabled.

For more information, see *Alignment fault* on page B4-14.

M, bit [0] MPU enable bit. This is a global enable bit for the MPU:

0 MPU disabled

1 MPU enabled.

For more information, see *Enabling and disabling the MPU* on page B4-5.

Reset value of the SCTLR

The SCTLR has a defined reset value that is IMPLEMENTATION DEFINED. There are different types of bit in the SCTLR:

- Some bits are defined as RAZ or RAO, and have the same value in all PMSAv7 implementations. Figure B4-6 on page B4-50 shows the values of these bits.
- Some bits are read-only and either:
  - — have an IMPLEMENTATION DEFINED value
  - — have a value that is determined by a configuration input signal.
- Some bits are read/write and either:
  - — reset to zero
  - — reset to an IMPLEMENTATION DEFINED value
  - — reset to a value that is determined by a configuration input signal.

Figure B4-6 on page B4-50 shows the reset value, or how the reset value is defined, for each bit of the SCTLR. It also shows the possible values of each half byte of the register.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ビット [6:3] RAO/SBOP

C、ビット[2] キャッシュイネーブル ビット。データキャッシュおよび統一キャッシュのグローバルイネーブル ビットです。

0 データキャッシュおよび統一キャッシュを非稼動にします。

1 データキャッシュおよび統一キャッシュを稼動します。

システムのメモリ階層のどのレベルにも、プロセッサからアクセス可能なデータまたは統一キャッシュが実装されていない場合、このビットは RAZ/WI です。

システムに、プロセッサからアクセス可能なデータまたは統一キャッシュが実装されている場合は、このビットを 0 にセットして、それらのキャッシュを非稼動にできる必要があります。

キャッシュの稼動の影響については、*キャッシュの稼動と非稼動* P. B2-8 で説明します。

A、ビット[1] アライメントビット。アライメントフォールト チェックのイネーブルビットです。

0 アライメントフォールト チェックを不可能にします。

1 アライメントフォールト チェックを可能にします。

詳細については、*アライメントフォールト* P. B4-14 を参照して下さい。

M、ビット[0] MPU イネーブルビット。MPU のグローバルイネーブル ビットです。

0 MPU を非稼動にします。

1 MPU を稼動します。

詳細については、*MPU の許可と禁止* P. B4-5 を参照して下さい。

SCTLR のリセット時の値

SCTLR のリセット時の値は実装定義です。SCTLR にはさまざまな種類のビットがあります。

- 一部のビットは RAZ または RAO として定義され、すべての PMSAv7 実装で同じ値です。これらのビットの値を、P. B4-50 の図 B4-6 に示します。
- 一部のビットは読み出し専用で、次のいずれかです。
  - – 実装定義の値を持つ。
  - – 構成入力信号によって決定される値を持つ。
- 一部のビットは読み出し / 書き込み可能で、次のいずれかです。
  - – 0 にリセットされる。
  - – 実装定義の値にリセットされる。
  - – 構成入力信号によって決定される値にリセットされる。

SCTLR の各ビットのリセット時の値、またはリセット時の値がどのように定義されるかを、P. B4-50 の図 B4-6 に示します。レジスタの各ハーフバイトの可能な値も示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

* Read-only bits, including RAZ and RAO bits.
(*) Can be RAZ. Otherwise read/write, resets to 0.
† Value is IMPLEMENTATION DEFINED.
(†) Can be read-only, with IMPLEMENTATION DEFINED value. Otherwise resets to 0.
‡ Value or reset value can depend on configuration input. Otherwise RAZ or resets to 0.

Figure B4-6 Reset value of the SCTLR, ARMv7-R (PMSAv7)

Accessing the SCTLR

To access SCTLR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c1, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c1,c0,0    ; Read CP15 System Control Register
MCR p15,0,<Rt>,c1,c0,0    ; Write CP15 System Control Register
```

Note

Additional configuration and control bits might be added to the SCTLR in future versions of the ARM architecture. ARM strongly recommends that software always uses a read, modify, write sequence to update the SCTLR. This prevents software modifying any bit that is currently unallocated, and minimizes the chance of the register update having undesired side effects.

B4.6.17 c1, IMPLEMENTATION DEFINED Auxiliary Control Register (ACTLR)

The Auxiliary Control Register, ACTLR, provides implementation-specific configuration and control options.

The ACTLR is:

- A 32-bit read/write register.
- Accessible only in privileged modes.

The contents of this register are IMPLEMENTATION DEFINED. ARMv7 requires this register to be privileged read/write accessible, even if an implementation has not created any control bits in this register.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

* RAZビットとRAOビットを含む読み出し専用ビット
(*) RAZにすることができる。それ以外は、読み出し/ 書き込み、0にリセット。
† 値は実装定義
(†) 実装定義の値を使用した読み出し専用にすることができる。それ以外は、0にリセット。
‡ 値またはリセット時の値は構成入力によって変えることができる。
それ以外は、RAZまたは0にリセット。

図 B4-6 SCTLR のリセット時の値、ARMv7-R (PMSAv7)

SCTLR へのアクセス

SCTLR にアクセスするには、`<opc1>` を 0、`<CRn>` を c1、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c1,c0,0    ; Read CP15 System Control Register
MCR p15,0,<Rt>,c1,c0,0    ; Write CP15 System Control Register
```

注

将来のバージョンの ARM アーキテクチャで、新しい構成ビットと制御ビットが SCTLR に追加される可能性があります。ソフトウェアで SCTLR を更新する場合は、常に読み出し – 変更 – 書き込みシーケンスを使用することを強くお勧めします。この方法を使用すれば、現時点で割り当てられていないビットがソフトウェアで変更されないため、不要な副作用を伴うレジスタ更新の機会が最小化されます。

B4.6.17 c1、実装定義の補助制御レジスタ (ACTLR)

補助制御レジスタ (ACTLR) は、実装固有の構成および制御のオプションを提供します。

ACTLR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能

このレジスタの内容は実装定義です。ARMv7 では、実装でこのレジスタにいずれの制御ビットも作成されていない場合でも、このレジスタを特権読み出し / 書き込みアクセス可能にする必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the ACTLR

To access the ACTLR you read or write the CP15 registers with <opc1> set to 0, <CRn> set to c1, <CRm> set to c0, and <opc2> set to 1. For example:

```
MRC p15,0,<Rt>,c1,c0,1    ; Read CP15 Auxiliary Control Register
MCR p15,0,<Rt>,c1,c0,1    ; Write CP15 Auxiliary Control Register
```

B4.6.18 c1, Coprocessor Access Control Register (CPACR)

The Coprocessor Access Control Register, CPACR, controls access to all coprocessors other than CP14 and CP15. It also enables software to check for the presence of coprocessors CP0 to CP13.

The CPACR:

- is a 32-bit read/write register
- is accessible only in privileged modes.
- has a defined reset value of 0.

The format of the CPACR is:

ASEDIS, bit[31]

Disable Advanced SIMD functionality:

0 This bit does not cause any instructions to be UNDEFINED.

1 All instruction encodings identified in the *Alphabetical list of instructions* on page A8-14 as being part of Advanced SIMD, but that are not VFPv3 instructions, are UNDEFINED.

On an implementation that:

- Implements VFP and does not implement Advanced SIMD, this bit is RAO/WI.
- Does not implement VFP or Advanced SIMD, this bit is UNK/SBZP.
- Implements both VFP and Advanced SIMD, it is IMPLEMENTATION DEFINED whether this bit is supported. If it is not supported it is RAZ/WI.

This bit resets to 0 if it is supported.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ACTLR へのアクセス

ACTLR にアクセスするには、<opc1> を 0、<CRn> を c1、<CRm> を c0、<opc2> を 1 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c1,c0,1    ; Read CP15 Auxiliary Control Register
MCR p15,0,<Rt>,c1,c0,1    ; Write CP15 Auxiliary Control Register
```

B4.6.18 c1、コプロセッサアクセス制御レジスタ (CPACR)

コプロセッサアクセス制御レジスタ (CPACR) は、CP14 と CP15 を除くすべてのコプロセッサへのアクセスを制御します。また、ソフトウェアでコプロセッサ CP0 ～ CP13 の存在のチェックを可能にします。

CPACR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- リセット時の値は 0 に定義されています。

CPACR のフォーマットは次のとおりです。

ASEDIS、ビット [31]

アドバンスト SIMD 機能の非稼動

0 このビットによって、命令が未定義になることはありません。

1 *命令のアルファベット順リスト* P. A8-14 ではアドバンスト SIMD の一部として識別されているが、VFPv3 命令ではないすべての命令エンコードは未定義です。

実装によって、このビットには次の相違があります。

- VFP が実装され、アドバンスト SIMD が実装されていない場合、このビットは RAO/WI です。
- VFP またはアドバンスト SIMD が実装されていない場合、このビットは UNK/SBZP です。
- VFP とアドバンスト SIMD の両方が実装されている場合、このビットがサポートされるかどうかは実装定義です。サポートされない場合、このビットは RAZ/WI です。

このビットがサポートされている場合は、0 にリセットされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D32DIS, bit[30]

Disable use of D16-D31 of the VFP register file:

0 This bit does not cause any instructions to be UNDEFINED.

1 All instruction encodings identified in the *Alphabetical list of instructions* on page A8-14 as being VFPv3 instructions are UNDEFINED if they access any of registers D16-D31.

If this bit is 1 when CPACR.ASEDIS == 0, the result is UNPREDICTABLE.

On an implementation that:

- Does not implement VFP, this bit is UNK/SBZP.
- Implements VFP and does not implement D16-D31, this bit is RAO/WI.
- Implements VFP and implements D16-D31, it is IMPLEMENTATION DEFINED whether this bit is supported. If it is not supported it is RAZ/WI.

This bit resets to 0 if it is supported.

Bits [29:28] Reserved. UNK/SBZP.

cp<n>, bits [2n+1, 2n], for n = 0 to 13

Defines the access rights for coprocessor n. The possible values of the field are:

00 Access denied. Any attempt to access the coprocessor generates an Undefined Instruction exception.

01 Privileged access only. Any attempt to access the coprocessor in User mode generates an Undefined Instruction exception.

10 Reserved. The effect of this value is UNPREDICTABLE.

11 Full access. The meaning of full access is defined by the appropriate coprocessor.

The value for a coprocessor that is not implemented is 00, access denied.

If more than one coprocessor is used to provide a set of functionality then having different values for the CPACR fields for those coprocessors can lead to UNPREDICTABLE behavior. An example where this must be considered is with the VFP extension, that uses CP10 and CP11.

Typically, an operating system uses this register to control coprocessor resource sharing among applications:

- Initially all applications are denied access to the shared coprocessor-based resources.
- When an application attempts to use a resource it results in an Undefined Instruction exception.
- The Undefined Instruction handler can then grant access to the resource by setting the appropriate field in the CPACR.

For details of how this register can be used to check for implemented coprocessors see *Access controls on CP0 to CP13* on page B1-63.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

D32DIS、ビット [30]

VFP レジスタファイルの D16 ～ D31 の使用を不可能にします。

0 このビットによって、命令が未定義になることはありません。

1 レジスタ D16 ～ D31 のいずれかにアクセスする場合、*命令のアルファベット順リスト* P. A8-14 で VFPv3 命令として識別されるすべての命令エンコードは未定義です。

このビットが 1 で、CPACR.ASEDIS == 0 の場合、結果は予測不能です。

実装によって、このビットには次の相違があります。

- VFP が実装されていない場合、このビットは UNK/SBZP です。
- VFP が実装され、D16 ～ D31 が実装されていない場合、このビットは RAO/WI です。
- VFP と D16 ～ D31 の両方が実装されている場合、このビットがサポートされるかどうかは実装定義です。サポートされない場合、このビットは RAZ/WI です。

このビットがサポートされている場合は、0 にリセットされます。

ビット [29:28] 予約。UNK/SBZP

cp<n>、ビット [2n+1、2n]、n = 0 ～ 13 の場合

コプロセッサ n のアクセス権限を定義します。このフィールドの可能な値は次のとおりです。

00 アクセス拒否。コプロセッサにアクセスを試みると、未定義命令例外が発生します。

01 特権アクセスのみ。ユーザモードでコプロセッサにアクセスを試みると、未定義命令例外が発生します。

10 予約。この値の影響は、予測不能です。

11 完全アクセス。完全アクセスの意味は、該当するコプロセッサによって定義されます。

実装されていないコプロセッサの値は、アクセス拒否を示す 00 です。

複数のコプロセッサを使用して一連の機能が提供されている場合は、それらのコプロセッサの CPACR フィールドに別々の値を設定すると、予測不能な動作につながる可能性があります。これを考慮しなければならない例は、CP10 と CP11 が使用される VFP 拡張機能を含む場合です。

通常、オペレーティングシステムでは、このレジスタを使用して、アプリケーション間で共有されるコプロセッサリソースが制御されます。

- 初期状態では、すべてのアプリケーションが、共有されているコプロセッサベースのリソースへのアクセスを拒否されます。
- アプリケーションがリソースの使用を試みると、未定義命令例外が発生します。
- 未定義命令ハンドラで、CPACR の該当するフィールドを設定することによって、リソースに対するアクセス権限を付与することができます。

このレジスタを使用して、実装されているコプロセッサをチェックする方法の詳細については、*CP0 ～ CP13 のアクセス制御* P. B1-63 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Sharing resources among applications requires a state saving mechanism. Two possibilities are:

- during a context switch, if the last executing process or thread had access rights to a coprocessor then the operating system saves the state of that coprocessor
- on receiving a request for access to a coprocessor, the operating system saves the old state for that coprocessor with the last process or thread that accessed it.

Accessing the CPACR

To access the CPACR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c1, `<CRm>` set to c0, and `<opc2>` set to 2. For example:

```
MRC p15,0,<Rt>,c1,c0,2    ; Read CP15 Coprocessor Access Control Register
MCR p15,0,<Rt>,c1,c0,2    ; Write CP15 Coprocessor Access Control Register
```

Normally, software uses a read, modify, write sequence to update the CPACR, to avoid unwanted changes to the access settings for other coprocessors.

B4.6.19 CP15 c2 and c3, Not used on a PMSA implementation

The CP15 c2 and c3 register encodings are not used on an ARMv7-R implementation, see *Unallocated CP15 encodings* on page B4-27.

B4.6.20 CP15 c4, Not used

The CP15 c4 register encodings are not used on an ARMv7 implementation, see *Unallocated CP15 encodings* on page B4-27.

B4.6.21 CP15 c5 and c6, Memory system fault registers

The CP15 c5 and c6 registers are used for memory system fault reporting. In addition, c6 provides the MPU Region registers. Figure B4-7 on page B4-54 shows the CP15 c5 and c6 registers.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アプリケーション間でリソースを共有するには、状態保存機構が必要です。次のような 2 種類の方法で実現が可能です。

- コンテキストスイッチ時に、最後に実行されていたプロセスまたはスレッドにコプロセッサへのアクセス権限が付与されている場合は、オペレーティングシステムでそのコプロセッサの状態を保存します。
- コプロセッサへのアクセス要求を受け取ったときに、オペレーティングシステムが、そのコプロセッサの古い状態と、そのコプロセッサにアクセスした最後のプロセスまたはスレッドを保存します。

CPACR へのアクセス

CPACR にアクセスするには、`<opc1>` を 0、`<CRn>` を c1、`<CRm>` を c0、`<opc2>` を 2 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c1,c0,2    ; Read CP15 Coprocessor Access Control Register
MCR p15,0,<Rt>,c1,c0,2    ; Write CP15 Coprocessor Access Control Register
```

通常は、ソフトウェアで読み出し - 変更 - 書き込みシーケンスを使用して CPACR を更新することによって、他のコプロセッサのアクセス設定に対する不要な変更を回避します。

B4.6.19 CP15 c2 および c3、PMSA 実装では未使用

CP15 c2 および c3 のレジスタエンコードは、ARMv7-R 実装では使用されません。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

B4.6.20 CP15 c4、未使用

CP15 c4 のレジスタエンコードは、ARMv7 実装では使用されません。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

B4.6.21 CP15 c5 および c6、メモリシステム フォールトレジスタ

CP15 c5 および c6 レジスタは、メモリシステム フォールトの報告に使用されます。さらに、c6 は MPU 領域レジスタを提供します。CP15 c5 および c6 レジスタを、P. B4-54 の図 B4-7 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure B4-7 CP15 c5 and c6 registers in a PMSA implementation

All CP15 c5 and c6 register encodings not shown in Figure B4-7 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B4-27.

The CP15 c5 and c6 registers are described in:

- *CP15 c5, Fault status registers*
- *CP15 c6, Fault Address registers* on page B4-57
- *CP15 c6, Memory region programming registers* on page B4-59.

Also, these registers are used to report information about debug exceptions. For details see *Effects of debug exceptions on CP15 registers and the DBGWFAR* on page C4-4.

B4.6.22 CP15 c5, Fault status registers

There are two fault status registers, in CP15 c5, and the architecture provides encodings for two additional IMPLEMENTATION DEFINED registers. Table B4-16 summarizes these registers.

Table B4-16 Fault status registers

Register name	Description
Data Fault Status Register (DFSR)	*c5, Data Fault Status Register (DFSR)* on page B4-55
Instruction Fault Status Register (IFSR)	*c5, Instruction Fault Status Register (IFSR)* on page B4-56
Auxiliary Data Fault Status Register (ADFSR)	*c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B4-56
Auxiliary Instruction Fault Status Register (AIFSR)	

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

図 B4-7 PMSA 実装の CP15 c5 および c6 レジスタ

図 B4-7 に示されていない CP15 c5 および c6 レジスタのエンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

CP15 c5 および c6 レジスタについては、次のセクションで説明します。

- *CP15 c5、フォールトステータス レジスタ*
- *CP15 c6、フォールトアドレス レジスタ* P. B4-57
- *CP15 c6、メモリ領域プログラミングレジスタ* P. B4-59

これらのレジスタは、デバッグ例外に関する情報の報告にも使用されます。詳細については、*CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響* P. C4-4 を参照して下さい。

B4.6.22 CP15 c5、フォールトステータス レジスタ

CP15 c5 には 2 つのフォールトステータス レジスタがあり、アーキテクチャでは、その他に 2 つの実装定義レジスタのエンコードが提供されます。これらのレジスタを、表 B4-16 に示します。

表 B4-16 フォールトステータス レジスタ

レジスタ名	説明
データフォールト ステータスレジスタ (DFSR)	*c5、データフォールト ステータスレジスタ (DFSR)* P. B4-55
命令フォールト ステータスレジスタ (IFSR)	*c5、命令フォールト ステータスレジスタ (IFSR)* P. B4-56
補助データフォールト ステータスレジスタ (ADFSR)	*c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B4-56
補助命令フォールト ステータスレジスタ (AIFSR)	

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Fault information is returned using the fault status registers and the fault address registers described in *CP15 c6, Fault Address registers* on page B4-57. For details of how these registers are used see *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18.

c5, Data Fault Status Register (DFSR)

The Data Fault Status Register, DFSR, holds status information about the last data fault.

The DFSR is:

- a 32-bit read/write register
- accessible only in privileged modes.

The format of the DFSR is:

Bits [31:13,9:4]

UNK/SBZP.

ExT, bit [12] External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

WnR, bit [11] Write not Read bit. Indicates whether the abort was caused by a write or a read access:

0 Abort caused by a read access

1 Abort caused by a write access.

For faults on CP15 cache maintenance operations this bit always returns a value of 1.

FS, bits [10,3:0]

Fault status bits. For the valid encodings of these bits in an ARMv7-R implementation with a PMSA, see Table B4-7 on page B4-20.

All encodings not shown in the table are reserved.

For information about using the DFSR see *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18.

Accessing the DFSR

To access the DFSR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c5, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c5,c0,0    ; Read CP15 Data Fault Status Register
MCR p15,0,<Rt>,c5,c0,0    ; Write CP15 Data Fault Status Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

フォールト情報は、フォールトステータス レジスタと、*CP15 c6、フォールトアドレス レジスタ* P. B4-57 で説明されているフォールトアドレス レジスタを使用して返されます。これらのレジスタの使用方法の詳細については、*PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4-18 を参照して下さい。

c5、データフォールト ステータスレジスタ (DFSR)

データフォールト ステータスレジスタ (DFSR) には、最後のデータフォールトに関するステータス情報が保持されます。

DFSR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能

DFSR のフォーマットは次のとおりです。

ビット [31:13、9:4]

UNK/SBZP

ExT、ビット[12] 外部アボートタイプ。このビットは、外部アボートの実装定義の分類を提供するために使用できます。

外部アボート以外のアボートでは、このビットは常に 0 を返します。

WnR、ビット[11] 書き込み / 読み出しビット。アボートが書き込みアクセスと読み出しアクセスのどちらで引き起こされたのかを示します。

0 読み出しアクセスで引き起こされたアボートです。

1 書き込みアクセスで引き起こされたアボートです。

CP15 キャッシュ保守操作でのフォールトの場合は、このビットが常に 1 の値を返します。

FS、ビット [10、3:0]

フォールトステータス ビット。PMSA を含む ARMv7-R 実装における、これらのビットの有効なエンコードについては、P. B4-20 の表 B4-7 を参照して下さい。

この表に示されていないエンコードはすべて予約されています。

DFSR の使用方法については、*PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4-18 を参照して下さい。

DFSR へのアクセス

DFSR にアクセスするには、`<opc1>` を 0、`<CRn>` を c5、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c5,c0,0    ; Read CP15 Data Fault Status Register
MCR p15,0,<Rt>,c5,c0,0    ; Write CP15 Data Fault Status Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c5, Instruction Fault Status Register (IFSR)

The Instruction Fault Status Register, IFSR, holds status information about the last instruction fault.

The IFSR is:

- a 32-bit read/write register
- accessible only in privileged modes.

The format of the IFSR is:

Bits [31:13,11,9:4]

UNK/SBZP.

ExT, bit [12] External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

FS, bits [10,3:0]

Fault status bits.

See Table B4-7 on page B4-20 for the valid encodings of these bits. All encodings not shown in the table are reserved.

For information about using the IFSR see *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18.

Accessing the IFSR

To access the IFSR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c5, `<CRm>` set to c0, and `<opc2>` set to 1. For example:

```
MRC p15,0,<Rt>,c5,c0,1    ; Read CP15 Instruction Fault Status Register
MCR p15,0,<Rt>,c5,c0,1    ; Write CP15 Instruction Fault Status Register
```

c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)

The Auxiliary Data Fault Status Register (ADFSR) and the Auxiliary Instruction Fault Status Register (AIFSR) enable the system to return additional IMPLEMENTATION DEFINED fault status information, see *Auxiliary Fault Status Registers* on page B4-21.

The ADFSR and AIFSR are:

- 32-bit read/write registers
- accessible only in privileged modes
- introduced in ARMv7.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c5、命令フォールト ステータスレジスタ (IFSR)

命令フォールト ステータスレジスタ (IFSR) には、最後の命令フォールトに関するステータス情報が保持されます。

IFSR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能

IFSR のフォーマットは次のとおりです。

ビット [31:13、11、9:4]

UNK/SBZP

ExT、ビット[12] 外部アボートタイプ。このビットは、外部アボートの実装定義の分類を提供するために使用できます。

外部アボート以外のアボートでは、このビットは常に 0 を返します。

FS、ビット [10、3:0]

フォールトステータス ビット。これらのビットの有効なエンコードについては、P. B4-20 の表 B4-7 を参照して下さい。この表に示されていないエンコードはすべて予約されています。

IFSR の使用方法については、*PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4-18 を参照して下さい。

IFSR へのアクセス

IFSR にアクセスするには、`<opc1>` を 0、`<CRn>` を c5、`<CRm>` を c0、`<opc2>` を 1 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c5,c0,1    ; Read CP15 Instruction Fault Status Register
MCR p15,0,<Rt>,c5,c0,1    ; Write CP15 Instruction Fault Status Register
```

c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)

補助データフォールト ステータスレジスタ (ADFSR) と補助命令フォールト ステータスレジスタ (AIFSR) は、システムが追加の実装定義のフォールトステータス情報を返すために使用できます。*補助フォールトステータス レジスタ* P. B4-21 を参照して下さい。

ADFSR と AIFSR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- ARMv7 で導入されました。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The formats of the ADFSR and AIFSR are IMPLEMENTATION DEFINED.

Accessing the ADFSR and AIFSR

To access the ADFSR or AIFSR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c5, `<CRm>` set to c1, and `<opc2>` set to:

- 0 for the ADFSR
- 1 for the AIFSR.

For example:

```
MRC p15,0,<Rt>,c5,c1,0    ; Read CP15 Auxiliary Data Fault Status Register
MCR p15,0,<Rt>,c5,c1,0    ; Write CP15 Auxiliary Data Fault Status Register
MRC p15,0,<Rt>,c5,c1,1    ; Read CP15 Auxiliary Instruction Fault Status Register
MCR p15,0,<Rt>,c5,c1,1    ; Write CP15 Auxiliary Instruction Fault Status Register
```

B4.6.23 CP15 c6, Fault Address registers

There are two Fault Address registers, in CP15 c6, as shown in Figure B4-7 on page B4-54. The two Fault Address registers complement the Fault Status registers, and are shown in Table B4-17.

Table B4-17 Fault address registers

Register name	Description
Data Fault Address Register (DFAR)	*c6, Data Fault Address Register (DFAR)*
Instruction Fault Address Register (IFAR)	*c6, Instruction Fault Address Register (IFAR)* on page B4-58

Note

Before ARMv7:

- The DFAR was called the Fault Address Register (FAR).
- The Watchpoint Fault Address Register (DBGWFAR) was implemented in CP15 c6 with `<opc2>` ==1. From ARMv7, the DBGWFAR is only implemented as a CP14 debug register, see *Watchpoint Fault Address Register (DBGWFAR)* on page C10-28.

Fault information is returned using the fault address registers and the fault status registers described in *CP15 c5, Fault status registers* on page B4-54. For details of how these registers are used see *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18.

c6, Data Fault Address Register (DFAR)

The Data Fault Address Register, DFAR, holds the faulting address that caused a synchronous Data Abort exception.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ADFSR と AIFSR のフォーマットは実装定義です。

ADFSR と AIFSR へのアクセス

ADFSR または AIFSR にアクセスするには、`<opc1>` を 0、`<CRn>` を c5、`<CRm>` を c1、`<opc2>` を次の値に設定して、CP15 レジスタを読み出し / 書き込みします。

- 0（ADFSR の場合）
- 1（AIFSR の場合）

次に例を示します。

```
MRC p15,0,<Rt>,c5,c1,0    ; Read CP15 Auxiliary Data Fault Status Register
MCR p15,0,<Rt>,c5,c1,0    ; Write CP15 Auxiliary Data Fault Status Register
MRC p15,0,<Rt>,c5,c1,1    ; Read CP15 Auxiliary Instruction Fault Status Register
MCR p15,0,<Rt>,c5,c1,1    ; Write CP15 Auxiliary Instruction Fault Status Register
```

B4.6.23 CP15 c6、フォールトアドレス レジスタ

P. B4-54 の図 B4-7 に示すように、CP15 c6 には、2 つのフォールトアドレス レジスタがあります。この 2 つのフォールトアドレス レジスタは、表 B4-17 に示すように、フォールトステータス レジスタを補完します。

表 B4-17 フォールトアドレス レジスタ

レジスタ名	説明
データフォールト アドレスレジスタ (DFAR)	*c6、データフォールト アドレスレジスタ (DFAR)*
命令フォールト アドレスレジスタ (IFAR)	*c6、命令フォールト アドレスレジスタ (IFAR)* P. B4-58

注

ARMv7 以前については、次の注意事項があります。

- DFAR はフォールトアドレス レジスタ (FAR) と呼ばれていました。
- ウォッチポイントフォールト アドレスレジスタ (DBGWFAR) は、`<opc2>` = 1 の CP15 c6 に実装されていました。ARMv7 およびそれ以降では、DBGWFAR は CP14 デバッグレジスタとしてのみ実装されます（*ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10-28 を参照）。

フォールト情報は、フォールトアドレス レジスタと、*CP15 c5、フォールトステータス レジスタ* P. B4-54 で説明されているフォールト ステータス レジスタを使用して返されます。これらのレジスタの使用方法の詳細については、*PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4-18 を参照して下さい。

c6、データフォールト アドレスレジスタ (DFAR)

データフォールト アドレスレジスタ (DFAR) には、同期データアボート例外を引き起こしたフォールトアドレスが保持されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The DFAR is:

- a 32-bit read/write register
- accessible only in privileged modes.

The format of the DFAR is:

31 … 0
Faulting address of synchronous Data Abort exception

For information about using the DFAR, including when the value in the DFAR is valid, see *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18.

A debugger can write to the DFAR to restore its value.

Accessing the DFAR

To access the DFAR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c6, `<CRm>` set to c0, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c6,c0,0    ; Read CP15 Data Fault Address Register
MCR p15,0,<Rt>,c6,c0,0    ; Write CP15 Data Fault Address Register
```

c6, Instruction Fault Address Register (IFAR)

The Instruction Fault Address Register, IFAR, holds the address of the faulting access that caused a synchronous Prefetch Abort exception.

The IFAR is:

- a 32-bit read/write register
- accessible only in privileged modes.

The format of the IFAR is:

31 … 0
Faulting address of synchronous Prefetch Abort exception

For information about using the IFAR, including when the value in the IFAR is valid, see *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18.

A debugger can write to the IFAR to restore its value.

Accessing the IFAR

To access the IFAR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c6, `<CRm>` set to c0, and `<opc2>` set to 2. For example:

```
MRC p15,0,<Rt>,c6,c0,2    ; Read CP15 Instruction Fault Address Register
MCR p15,0,<Rt>,c6,c0,2    ; Write CP15 Instruction Fault Address Register
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

DFAR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能

DFAR のフォーマットは次のとおりです。

31 … 0
同期データアボート例外のフォールトアドレス

DFAR の使用方法と、DFAR 内の値が有効になるタイミングについては、*PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4-18 を参照して下さい。

デバッガは、DFAR に書き込みを行い、その値を復元することができます。

DFAR へのアクセス

DFAR にアクセスするには、`<opc1>` を 0、`<CRn>` を c6、`<CRm>` を c0、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c6,c0,0    ; Read CP15 Data Fault Address Register
MCR p15,0,<Rt>,c6,c0,0    ; Write CP15 Data Fault Address Register
```

c6、命令フォールト アドレスレジスタ (IFAR)

命令フォールト アドレスレジスタ (IFAR) には、同期プリフェッチアボート例外を引き起こしたフォールトアクセスのアドレスが保持されます。

IFAR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能

IFAR のフォーマットは次のとおりです。

31 … 0
同期プリフェッチアボート例外のフォールトアドレス

IFAR の使用方法と、IFAR 内の値が有効になるタイミングについては、*PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4-18 を参照して下さい。

デバッガは、IFAR に書き込みを行い、その値を復元することができます。

IFAR へのアクセス

IFAR にアクセスするには、`<opc1>` を 0、`<CRn>` を c6、`<CRm>` を c0、`<opc2>` を 2 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c6,c0,2    ; Read CP15 Instruction Fault Address Register
MCR p15,0,<Rt>,c6,c0,2    ; Write CP15 Instruction Fault Address Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.24 CP15 c6, Memory region programming registers

When the PMSA is implemented, a number of registers in CP15 c6 are used to configure the MPU memory regions. There are three registers for each memory region supported by the MPU:

- A Base Address Register, that defined the start address of the region in the memory map.
- A Region Size and Enable Register, that:
  - — has a single enable bit for the region
  - — defines the size of the region
  - — has a disable bit for each of the eight subregions in the region.
- A Region Access Control Register that defines the memory access attributes for the region.

The multiple copies of these registers are mapped onto three or six registers in CP15 c6, and another register is used to select the current memory region. The mapping of the region registers onto the CP15 registers depends on whether the MPU implements a unified memory map, or separate Instruction and Data memory maps:

Separate Instruction and Data memory maps

The multiple copies of the registers that describe each memory region map onto six CP15 registers.

For the memory regions in the Instruction memory map:

- the multiple Region Base Address Registers map onto the Instruction Region Base Address Register, IRBAR
- the multiple Region Size and Enable Registers map onto the Instruction Region Size and Enable Register, IRSR
- the multiple Region Access Control Registers map onto the Instruction Region Access Control Register, IRACR.

For the memory regions in the Data memory map:

- the multiple Region Base Address Registers map onto the Data Region Base Address Register, DRBAR
- the multiple Region Size and Enable Registers map onto the Data Region Size and Enable Register, DRSR
- the multiple Region Access Control Registers map onto the Data Region Access Control Register, DRACR.

The value in the RGNR is the index value for both the instruction region and the data region registers, see *c6, MPU Region Number Register (RGNR)* on page B4-66. The RGNR value indicates the current memory region for both the instruction and the data memory maps. However, a particular value might not be valid for both memory maps.

Unified memory maps

The multiple copies of the registers that describe each memory region map onto three CP15 registers:

- the multiple Region Base Address Registers map onto the Data Region Base Address Register, DRBAR

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.24 CP15 c6、メモリ領域プログラミングレジスタ

PMSA が実装されている場合は、CP15 c6 の複数のレジスタが、MPU メモリ領域の構成に使用されます。MPU でサポートされているメモリ領域ごとに、次の 3 つのレジスタが存在します。

- メモリマップ内の領域の開始アドレスを定義するベースアドレス レジスタ
- 次の条件を満たす、領域サイズおよびイネーブルレジスタ
  - – 領域について単一のイネーブルビットが存在する
  - – 領域のサイズを定義する
  - – 領域の 8 つのサブ領域ごとに、ディセーブルビットが存在する
- 領域のメモリアクセス属性を定義する、領域アクセス制御レジスタ

これらのレジスタの複数のコピーが、CP15 c6 内の 3 つまたは 6 つのレジスタにマッピングされ、別のレジスタが現在のメモリ領域を選択するために使用されます。CP15 レジスタに対する領域レジスタのマッピングは、MPU で、統一メモリマップと、独立した命令およびデータ メモリ マップのどちらが実装されているかによって異なります。

独立した命令およびデータメモリマップ

各メモリ領域マップを記述するレジスタの複数のコピーが、6 つの CP15 レジスタにマッピングされます。

命令メモリマップのメモリ領域の場合、次の規則が適用されます。

- 複数の領域ベースアドレス レジスタが、命令領域ベースアドレス レジスタ (IRBAR) にマッピングされます。
- 複数の領域サイズおよびイネーブルレジスタが、命令領域サイズおよびイネーブルレジスタ (IRSR) にマッピングされます。
- 複数の領域アクセス制御レジスタが、命令領域アクセス制御レジスタ (IRACR) にマッピングされます。

データ メモリマップのメモリ領域の場合、次の規則が適用されます。

- 複数の領域ベースアドレス レジスタが、データ領域ベースアドレス レジスタ (DRBAR) にマッピングされます。
- 複数の領域サイズおよびイネーブルレジスタが、データ領域サイズおよびイネーブルレジスタ (DRSR) にマッピングされます。
- 複数の領域アクセス制御レジスタが、データ領域アクセス制御レジスタ (DRACR) にマッピングされます。

RGNR の値は、命令領域レジスタとデータ領域レジスタの両方に関するインデクス値です。*c6、MPU 領域番号レジスタ (RGNR)* P. B4-66 を参照して下さい。
RGNR の値は、命令メモリマップとデータ メモリマップの両方について、現在のメモリ領域を示し ます。ただし、特定の値が両方のメモリマップに対して有効でない場合があります。

統一メモリマップ

各メモリ領域マップを示すレジスタの複数のコピーが、3 つの CP15 レジスタにマッピングされます。

- 複数の領域ベースアドレス レジスタが、データ領域ベースアドレス レジスタ (DRBAR) にマッピングされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- the multiple Region Size and Enable Registers map onto the Data Region Size and Enable Register, DRSR
- the multiple Region Access Control Registers map onto the Data Region Access Control Register, DRACR.

The IRBAR, IRSR, and IRACR are not implemented.

The value in the RGNR is the index value for the data region registers, see *c6, MPU Region Number Register (RGNR)* on page B4-66. Its value indicates the current memory region in the unified memory map.

The read-only MPUIR indicates:

- whether the MPU implements separate Instruction and Data address maps, or a Unified address map
- the number of Data or Unified regions the MPU supports
- if separate Instruction and Data address maps are implemented, the number of Instruction regions the MPU supports.

For more information, see *c0, MPU Type Register (MPUIR)* on page B4-36.

Table B4-18 summarizes the CP15 registers that are used to program the MPU memory regions, and gives references to the full descriptions of these registers.

Table B4-18 MPU Memory Region Programming Registers

Register name	Description
Data or Unified Region Base Address	*c6, Data Region Base Address Register (DRBAR)*
Instruction Region Base Address[a]	*c6, Instruction Region Base Address Register (IRBAR)* on page B4-61[a]
Data or Unified Region Size and Enable	*c6, Data Region Size and Enable Register (DRSR)* on page B4-62
Instruction Region Size and Enable[a]	*c6, Instruction Region Size and Enable Register (IRSR)* on page B4-63[a]
Data or Unified Region Access Control	*c6, Data Region Access Control Register (DRACR)* on page B4-64
Instruction Region Access Control[a]	*c6, Instruction Region Access Control Register (IRACR)* on page B4-65[a]
MPU Region Number	*c6, MPU Region Number Register (RGNR)* on page B4-66

a. These registers are implemented only if the MPU implements separate Instruction and Data memory maps.

c6, Data Region Base Address Register (DRBAR)

The Data Region Base Address Register, DRBAR, indicates the base address of the current memory region in the data or unified address map. The base address must be aligned to the region size. The current memory region is selected by the value held in the RGNR, see *c6, MPU Region Number Register (RGNR)* on page B4-66.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- 複数の領域サイズおよびイネーブルレジスタが、データ領域サイズおよびイネーブルレジスタ (DRSR) にマッピングされます。
- 複数の領域アクセス制御レジスタが、データ領域アクセス制御レジスタ (DRACR) にマッピングされます。

IRBAR、IRSR、IRACR は実装されていません。

RGNR の値は、データ領域レジスタに関するインデクス値です。*c6、MPU 領域番号レジスタ (RGNR)* P. B4-66 を参照して下さい。この値は、統一メモリマップの現在のメモリ領域を示します。

読み出し専用 MPUIR は、次の情報を示します。

- MPU で、独立した命令およびデータ アドレスマップと、統一アドレスマップのどちらが実装されているか。
- MPU でサポートされるデータ領域または統一領域の数
- 独立した命令およびデータ アドレスマップが実装されている場合に、MPU でサポートされる命令領域の数

詳細については、*c0、MPU タイプレジスタ (MPUIR)* P. B4-36 を参照して下さい。

MPU メモリ領域をプログラムするために使用される CP15 レジスタの概要と、これらのレジスタの完全な詳細への参照を、表 B4-18 に示します。

表 B4-18 MPU メモリ領域プログラミングレジスタ

レジスタ名	説明
データまたは統一領域ベースアドレス	*c6、データ領域ベースアドレス レジスタ (DRBAR)*
命令領域ベースアドレス[a]	*c6、命令領域ベースアドレス レジスタ (IRBAR)* P. B4-61[a]
データまたは統一領域サイズおよびイネーブル	*c6、データ領域サイズおよびイネーブルレジスタ (DRSR)* P. B4-62
命令領域サイズおよびイネーブル[a]	*c6、命令領域サイズおよびイネーブルレジスタ (IRSR)* P. B4-63[a]
データまたは統一領域アクセス制御	*c6、データ領域アクセス制御レジスタ (DRACR)* P. B4-64
命令領域アクセス制御[a]	*c6、命令領域アクセス制御レジスタ (IRACR)* P. B4-65[a]
MPU 領域番号	*c6、MPU 領域番号レジスタ (RGNR)* P. B4-66

a. これらのレジスタは、MPU で、独立した命令およびデータ メモリマップが実装されている場合にのみ実装されます。

c6、データ領域ベースアドレス レジスタ (DRBAR)

データ領域ベースアドレス レジスタ (DRBAR) は、データまたは統一アドレスマップの現在のメモリ領域のベースアドレスを示します。このベースアドレスは、領域サイズにアラインしている必要があります。現在のメモリ領域は、RGNR に保持された値によって選択されます。*c6、MPU 領域番号レジスタ (RGNR)* P. B4-66 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The DRBAR is:

- a 32-bit read/write register
- accessible only in privileged modes.

The format of the DRBAR is:

31	2 1 0
Region Base Address	(0) (0)

Region Base Address, bits [31:2]

The Base Address for the region, in the Data or Unified address map. The region referenced is selected by the RGNR

Bit [1:0] UNK/SBZP.

The DRBAR can be used to find the size of the supported physical address space for the Data or Unified memory map, see *Finding the minimum supported region size* on page B4-7.

Accessing the DRBAR

To access the DRBAR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c6, `<CRm>` set to c1, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c6,c1,0    ; Read  CP15 Data Region Base Address Register
MCR p15,0,<Rt>,c6,c1,0    ; Write CP15 Data Region Base Address Register
```

c6, Instruction Region Base Address Register (IRBAR)

The Instruction Region Base Address Register, IRBAR, indicates the base address of the current memory region in the Instruction address map. The base address must be aligned to the region size. The current memory region is selected by the value held in the RGNR, see *c6, MPU Region Number Register (RGNR)* on page B4-66.

The IRBAR is:

- a 32-bit read/write register
- accessible only in privileged modes.
- implemented only when the PMSA implements separate instruction and data memory maps.

The format of the IRBAR is identical to the DRBAR, see *c6, Data Region Base Address Register (DRBAR)* on page B4-60.

The IRBAR can be used to find the minimum region size supported by the implementation, see *Finding the minimum supported region size* on page B4-7.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DRBAR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能

DRBAR のフォーマットは次のとおりです。

31	2 1 0
領域ベースアドレス	(0) (0)

領域ベースアドレス、ビット [31:2]

データまたは統一アドレスマップの領域のベースアドレス。参照される領域は、RGNR によって選択されます。

ビット [1:0] UNK/SBZP

DRBAR は、データまたは統一メモリマップについて、サポートされている物理アドレス空間のサイズを特定するために使用されます。*サポートされている最小領域サイズの判定* P. B4-7 を参照して下さい。

DRBAR へのアクセス

DRBAR にアクセスするには、`<opc1>` を 0、`<CRn>` を c6、`<CRm>` を c1、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c6,c1,0    ; Read  CP15 Data Region Base Address Register
MCR p15,0,<Rt>,c6,c1,0    ; Write CP15 Data Region Base Address Register
```

c6、命令領域ベースアドレス レジスタ (IRBAR)

命令領域ベースアドレス レジスタ (IRBAR) は、命令アドレスマップの現在のメモリ領域のベースアドレスを示します。このベースアドレスは、領域サイズにアラインしている必要があります。現在のメモリ領域は、RGNR に保持されている値によって選択されます。*c6、MPU 領域番号レジスタ (RGNR)* P. B4-66 を参照して下さい。

IRBAR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- PMSA が独立した命令およびデータ メモリマップを実装している場合にのみ実装されます。

IRBAR のフォーマットは、DRBAR と同じです。*c6、データ領域ベースアドレス レジスタ (DRBAR)* P. B4-60 を参照して下さい。

IRBAR を使用すると、実装でサポートされている最小領域サイズを特定できます。*サポートされている最小領域サイズの判定* P. B4-7 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the IRBAR

To access the IRBAR you read or write the CP15 registers with <opc1> set to 0, <CRn> set to c6, <CRm> set to c1, and <opc2> set to 1. For example:

```
MRC p15,0,<Rt>,c6,c1,1    ; Read  CP15 Instruction Region Base Address Register
MCR p15,0,<Rt>,c6,c1,1    ; Write CP15 Instruction Region Base Address Register
```

c6, Data Region Size and Enable Register (DRSR)

The Data Region Size and Enable Register, DRSR, indicates the size of the current memory region in the data or unified address map, and can be used to enable or disable:

- the entire region
- each of the eight subregions, if the region is enabled.

The current memory region is selected by the value held in the RGNR see *c6, MPU Region Number Register (RGNR)* on page B4-66.

The DRSR:

- is a 32-bit read/write register
- is accessible only in privileged modes.
- has a defined reset value of 0.

The format of the DRSR is:

Bit [31:16,7:6]

UNK/SBZP.

SnD, bit [n+8], for values of n from 0 to 7

Subregion disable bit for region n. Indicates whether the subregion is part of this region:

0 Subregion is part of this region

1 Subregion disabled. The subregion is not part of this region.

The region is divided into exactly eight equal sized subregions. Subregion 0 is the subregion at the least significant address. For more information, see *Subregions* on page B4-3.

If the size of this region, indicated by the RSize field, is less than 256 bytes then the SnD fields are not defined, and register bits [15:8] are UNK/SBZP.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

IRBAR へのアクセス

IRBAR にアクセスするには、<opc1> を 0、<CRn> を c6、<CRm> を c1、<opc2> を 1 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c6,c1,1    ; Read  CP15 Instruction Region Base Address Register
MCR p15,0,<Rt>,c6,c1,1    ; Write CP15 Instruction Region Base Address Register
```

c6、データ領域サイズおよびイネーブルレジスタ (DRSR)

データ領域サイズおよびイネーブルレジスタ (DRSR) は、データまたは統一アドレスマップの現在のメモリ領域のサイズを示します。また、次のような領域を許可または禁止するためにも使用できます。

- 領域全体
- 稼動している 8 つのサブ領域のそれぞれ

現在のメモリ領域は、RGNR に保持されている値によって選択されます。*c6、MPU 領域番号レジスタ (RGNR)* P. B4- 66 を参照して下さい。

DRSR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- リセット時の値は 0 に定義されています。

DRSR のフォーマットは次のとおりです。

ビット [31:16、7:6]

UNK/SBZP

SnD、ビット [n + 8]、n の値が 0 ～ 7 の場合

領域 n のサブ領域ディセーブルビット。サブ領域がその領域の一部であるかどうかを示します。

0 サブ領域は、その領域の一部です。

1 サブ領域が非稼動です。サブ領域は、その領域の一部ではありません。

領域は、サイズの等しい 8 つのサブ領域に分割されます。サブ領域 0 は、最下位アドレスにあるサブ領域です。詳細については、*サブ領域* P. B4- 3 を参照して下さい。

RSize フィールドで示されるその領域のサイズが 256 バイト未満の場合は、SnD フィールドは定義されず、レジスタのビット [15:8] は UNK/SBZP です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

RSize, bits [5:1]

Region Size field. Indicates the size of the current memory region:

- A value of 0 is not permitted, this value is reserved and UNPREDICTABLE.
- If N is the value in this field, the region size is 2^{N+1} bytes.

En, bit [0] Enable bit for the region:

0 Region is disabled

1 Region is enabled.

Because this register resets to zero, all memory regions are disabled on reset.

All memory regions must be enabled before they are used.

The minimum region size supported is IMPLEMENTATION DEFINED, but if the memory system implementation includes a cache, ARM strongly recommends that the minimum region size is a multiple of the cache line length. This prevents cache attributes changing mid-way through a cache line.

Behavior is UNPREDICTABLE if you:

- write a region size that is outside the range supported by the implementation
- access this register when the RGNR does not point to a valid region in the MPU Data or Unified address map.

Accessing the DRSR

To access the DRSR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c6, `<CRm>` set to c1, and `<opc2>` set to 2. For example:

```
MRC p15,0,<Rt>,c6,c1,2    ; Read  CP15 Data Region Size and Enable Register
MCR p15,0,<Rt>,c6,c1,2    ; Write CP15 Data Region Size and Enable Register
```

c6, Instruction Region Size and Enable Register (IRSR)

The Instruction Region Size and Enable Register, IRSR, indicates the size of the current memory region in the instruction address map, and to enable or disable:

- the entire region
- each of the eight subregions, if the region is enabled.

The current memory region is selected by the value held in the RGNR, see *c6, MPU Region Number Register (RGNR)* on page B4-66.

The IRSR:

- is a 32-bit read/write register
- is accessible only in privileged modes
- has a defined reset value of 0.
- is implemented only when the PMSA implements separate instruction and data memory maps.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

RSize、ビット [5:1]

領域サイズフィールド。現在のメモリ領域のサイズを示します。

- 0 の値は許可されません。この値は予約されており、予測不能です。
- このフィールドの値が N の場合、領域サイズは 2^{N+1} バイトです。

En、ビット [0] 領域のイネーブルビット。

0 領域は稼動していません。

1 領域は稼動しています。

このレジスタは 0 にリセットされるため、リセット時にはすべてのメモリ領域が非稼動になります。

すべてのメモリ領域は、使用する前に稼動する必要があります。

サポートされる最小の領域サイズは実装定義ですが、メモリシステム実装にキャッシュが含まれている場合は、最小の領域サイズをキャッシュライン長の倍数にすることを強くお勧めします。これによって、キャッシュラインの途中でキャッシュ属性が変化することが防止されます。

次の場合、動作は予測不能です。

- 実装でサポートされている範囲を超えた領域サイズを書き込む。
- RGNR が MPU のデータまたは統一アドレスマップ内の有効な領域を指していないときに、このレジスタにアクセスする。

DRSR へのアクセス

DRSR にアクセスするには、`<opc1>` を 0、`<CRn>` を c6、`<CRm>` を c1、`<opc2>` を 2 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c6,c1,2    ; Read  CP15 Data Region Size and Enable Register
MCR p15,0,<Rt>,c6,c1,2    ; Write CP15 Data Region Size and Enable Register
```

c6、命令領域サイズおよびイネーブルレジスタ (IRSR)

命令領域サイズおよびイネーブルレジスタ (IRSR) は、命令アドレスマップの現在のメモリ領域のサイズを示します。また、次の領域を稼動または非稼動にするために使用することもできます。

- 領域全体
- 許可されている 8 つのサブ領域のそれぞれ

現在のメモリ領域は、RGNR に保持されている値によって選択されます。*c6、MPU 領域番号レジスタ (RGNR)* P. B4- 66 を参照して下さい。

IRSR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- リセット時の値は 0 に定義されています。
- PMSA が独立した命令およびデータ メモリマップを実装している場合にのみ実装されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the IRSR is identical to the DRSR, see *c6, Data Region Size and Enable Register (DRSR)* on page B4-62.

All memory regions must be enabled before they are used.

The minimum region size supported is IMPLEMENTATION DEFINED, but if the memory system implementation includes an instruction cache, ARM strongly recommends that the minimum region size is a multiple of the instruction cache line length. This prevents cache attributes changing mid-way through a cache line.

Behavior is UNPREDICTABLE if you:

- write a region size that is outside the range supported by the implementation
- access this register when the RGNR does not point to a valid region in the MPU instruction address map.

Accessing the IRSR

To access the IRSR you read or write the CP15 registers with <opc1> set to 0, <CRn> set to c6, <CRm> set to c1, and <opc2> set to 3. For example:

```
MRC p15,0,<Rt>,c6,c1,3    ; Read  CP15 Instruction Region Size and Enable Register
MCR p15,0,<Rt>,c6,c1,3    ; Write CP15 Instruction Region Size and Enable Register
```

c6, Data Region Access Control Register (DRACR)

The Data Region Access Control Register, DRACR, defines the memory attributes for the current memory region in the data or unified address map.

The current memory region is selected by the value held in the RGNR, see *c6, MPU Region Number Register (RGNR)* on page B4-66.

The DRACR is:

- a 32-bit read/write register
- accessible only in privileged modes.

The format of the DRACR is:

31 … 13	12	11	10 … 8	7	6	5 … 3	2	1	0
UNK/SBZP	XN	(0)	AP [2:0]	(0)	(0)	TEX [2:0]	S	C	B

Bit [31:13,11,7:6]

UNK/SBZP.

XN, bit [12] Execute Never bit. Indicates whether instructions can be fetched from this region:

0 region can contain executable code

1 region is an Execute never region, and any attempt to execute an instruction from the region results in a Permission fault.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

IRSR のフォーマットは、DRSR と同じです。*c6、データ領域サイズおよびイネーブルレジスタ (DRSR)* P. B4- 62 を参照して下さい。

すべてのメモリ領域は、使用する前に稼動する必要があります。

サポートされる最小の領域サイズは実装定義ですが、メモリシステム実装に命令キャッシュが含まれている場合は、最小の領域サイズをキャッシュライン長の倍数にすることを強くお勧めします。これによって、キャッシュラインの途中でキャッシュ属性が変化することが防止されます。

次の場合、動作は予測不能です。

- 実装でサポートされている範囲を超えた領域サイズを書き込む。
- RGNR が MPU 命令アドレスマップの有効な領域を指していないときに、このレジスタにアクセスする。

IRSR へのアクセス

IRSR にアクセスするには、<opc1> を 0、<CRn> を c6、<CRm> を c1、<opc2> を 3 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c6,c1,3    ; Read  CP15 Instruction Region Size and Enable Register
MCR p15,0,<Rt>,c6,c1,3    ; Write CP15 Instruction Region Size and Enable Register
```

c6、データ領域アクセス制御レジスタ (DRACR)

データ領域アクセス制御レジスタ (DRACR) は、データまたは統一アドレスマップの現在のメモリ領域に関するメモリ属性を定義します。

現在のメモリ領域は、RGNR に保持されている値によって選択されます。*c6、MPU 領域番号レジスタ (RGNR)* P. B4- 66 を参照して下さい。

DRACR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能

DRACR のフォーマットは次のとおりです。

31 … 13	12	11	10 … 8	7	6	5 … 3	2	1	0
UNK/SBZP	XN	(0)	AP[2:0]	(0)	(0)	TEX[2:0]	S	C	B

ビット [31:13、11、7:6]

UNK/SBZP

XN、ビット [12] 実行不可ビット。この領域から命令をフェッチできるかどうかを示します。

0 領域に実行可能コードを含めることができます。

1 領域は実行不可領域で、その領域から命令の実行を試みると、アクセス許可フォールトが発生します。

MPU で独立した命令およびデータ メモリマップが実装されている場合、このビットは UNK/SBZ です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If the MPU implements separate Instruction and Data memory maps this bit is UNK/SBZ

For more information, see *The Execute Never (XN) attribute and instruction prefetching* on page B4-10.

AP[2:0], bits [10:8]

Access Permissions field. Indicates the read and write access permissions for unprivileged and privileged accesses to the memory region.

For more information, see *Access permissions* on page B4-9.

TEX[2:0], C, B, bits [5:3,1:0]

Memory access attributes. For more information, see *C, B, and TEX[2:0] encodings* on page B4-11.

S, bit [2] Shareable bit, for Normal memory regions:

0 If region is Normal memory, memory is Non-shareable

1 If region is Normal memory, memory is Shareable.

The value of this bit is ignored if the region is not Normal memory.

If you access this register when the RGNR does not point to a valid region in the MPU data or unified address map, the result is UNPREDICTABLE.

Accessing the DRACR

To access the DRACR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c6, `<CRm>` set to c1, and `<opc2>` set to 4. For example:

```
MRC p15,0,<Rt>,c6,c1,4    ; Read  CP15 Data Region Access Control Register
MCR p15,0,<Rt>,c6,c1,4    ; Write CP15 Data Region Access Control Register
```

c6, Instruction Region Access Control Register (IRACR)

The Instruction Region Access Control Register, IRACR, defines the memory attributes for the current memory region in the instruction address map, when the MPU implements separate data and instruction address maps.

The current memory region is selected by the value held in the RGNR, see *c6, MPU Region Number Register (RGNR)* on page B4-66.

The IRACR is:

- a 32-bit read/write register
- accessible only in privileged modes
- implemented only when the PMSA implements separate instruction and data memory maps.

The format of the IRACR is identical to the DRACR, see *c6, Data Region Access Control Register (DRACR)* on page B4-64.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

詳細については、*実行不可 (XN) 属性と命令のプリフェッチ* P. B4-10 を参照して下さい。

AP[2:0]、ビット [10:8]

アクセス許可フィールド。メモリ領域への非特権アクセスと特権アクセスについて、読み出しおよび書き込みのアクセス許可を示します。

詳細については、*アクセス許可* P. B4-9 を参照して下さい。

TEX[2:0]、C、B、ビット [5:3、1:0]

メモリアクセス属性。詳細については、*C*、*B*、*TEX[2:0] のエンコード* P. B4-11 を参照して下さい。

S、ビット [2] ノーマルメモリ領域の場合の共有可能ビット

0 領域がノーマルメモリの場合、メモリは共有不可です。

1 領域がノーマルメモリの場合、メモリは共有可能です。

このビットの値は、領域がノーマルメモリでない場合は無視されます。

RGNR が MPU のデータまたは統一アドレスマップの有効な領域を指していないときに、このレジスタにアクセスした場合、結果は予測不能です。

DRACR へのアクセス

DRACR にアクセスするには、`<opc1>` を 0、`<CRn>` を c6、`<CRm>` を c1、`<opc2>` を 4 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c6,c1,4    ; Read  CP15 Data Region Access Control Register
MCR p15,0,<Rt>,c6,c1,4    ; Write CP15 Data Region Access Control Register
```

c6、命令領域アクセス制御レジスタ (IRACR)

命令領域アクセス制御レジスタ (IRACR) は、MPU で独立したデータおよび命令アドレスマップが実装されている場合の、命令アドレスマップの現在のメモリ領域に関するメモリ属性を定義します。

現在のメモリ領域は、RGNR に保持されている値によって選択されます。*c6、MPU 領域番号レジスタ (RGNR)* P. B4-66 を参照して下さい。

IRACR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- PMSA が独立した命令およびデータ メモリマップを実装している場合にのみ実装されます。

IRACR のフォーマットは、DRACR と同じです。*c6、データ領域アクセス制御レジスタ (DRACR)* P. B4-64 を参照して下さい。

注

IRACR の XN ビット（ビット [12]）は常に有効です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

The XN bit, bit [12], is always valid in the IRACR.

If you access this register when the RGNR does not point to a valid region in the MPU instruction address map, the result is UNPREDICTABLE.

Accessing the IRACR

To access the IRACR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c6, `<CRm>` set to c1, and `<opc2>` set to 5. For example:

```
MRC p15,0,<Rt>,c6,c1,5    ; Read  CP15 Instruction Region Access Control Register
MCR p15,0,<Rt>,c6,c1,5    ; Write CP15 Instruction Region Access Control Register
```

c6, MPU Region Number Register (RGNR)

The MPU Region Number Register, RGNR, defines the current memory region in:

- the MPU data or unified address map
- the MPU instruction address map, if the MPU implements separate data and instruction address maps.

The value in the RGNR identifies the memory region description accessed by the Region Base Address, Size and Enable, and Access Control Registers.

Note

There is only a single MPU Region Number Register. When the MPU implements separate data and instruction address maps, the current region number is always identical for both address maps. This might mean that the current region number is valid for one address map but invalid for the other map.

The RGNR is:

- a 32-bit read/write register
- accessible only in privileged modes.

The format of the RGNR is:

31 … N	N-1 … 0
UNK/SBZP	Region

Bit [31:N] UNK/SBZP.

Region, bits [N-1:0]

The number of the current region in the Data or Unified address map, and in the Instruction address map if the MPU implements separate Data and Instruction address maps.

The value of N is Log_2(Number of regions supported) rounded up to an integer.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

RGNR が MPU 命令アドレスマップの有効な領域を指していないときに、このレジスタにアクセスした場合、結果は予測不能です。

IRACR へのアクセス

IRACR にアクセスするには、`<opc1>` を 0、`<CRn>` を c6、`<CRm>` を c1、`<opc2>` を 5 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c6,c1,5    ; Read  CP15 Instruction Region Access Control Register
MCR p15,0,<Rt>,c6,c1,5    ; Write CP15 Instruction Region Access Control Register
```

c6、MPU 領域番号レジスタ (RGNR)

MPU 領域番号レジスタ (RGNR) では、次のマップに含まれる現在のメモリ領域が定義されます。

- MPU データまたは統一アドレスマップ
- MPU で独立したデータおよび命令メモリマップが実装されている場合の、MPU 命令アドレスマップ

RGNR の値によって、領域ベースアドレス、サイズおよびイネーブル、およびアクセス制御レジスタからアクセスされるメモリ領域の記述が識別されます。

注

MPU 領域番号レジスタは 1 つしかありません。MPU で独立したデータおよび命令アドレスマップが実装されている場合は、現在の領域番号は常に両方のアドレスマップで同じです。これは、現在の領域番号があるアドレスマップに対して有効であるが、別のマップに対して無効である場合があることを意味します。

RGNR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能

RGNR のフォーマットは次のとおりです。

31 … N	N − 1 … 0
UNK/SBZP	領域

ビット [31:N] UNK/SBZP

領域、ビット [N − 1:0]

データまたは統一アドレスマップ、および MPU で独立したデータおよび命令アドレスマップが実装されている場合の命令アドレスマップの、現在の領域番号。

N の値は、Log_2(サポートされている領域数) を、より大きい次の整数に丸めた値です。

メモリ領域の番号付けは 0 から始まり、サポートされている領域数 − 1 まで増加します。

サポートされているメモリ領域数以上の値をこのレジスタに書き込んだ場合、結果は予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996−1998, 2000, 2004−2008 ARM Limited. All rights reserved.

Memory region numbering starts at 0 and goes up to one less than the number of regions supported.

Writing a value to this register that is greater than or equal to the number of memory regions supported has UNPREDICTABLE results.

In the context of the RGNR description, when the MPU implements separate Data and Instruction address maps the Number of memory regions supported is the greater of:

- number of Data memory regions supported
- number of Instruction memory regions supported.

Accessing the RGNR

To access the RGNR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c6, `<CRm>` set to c2, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c6,c2,0    ; Read  CP15 MPU Region Number Register
MCR p15,0,<Rt>,c6,c2,0    ; Write CP15 MPU Region Number Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

RGNR 記述の観点では、MPU で独立したデータおよび命令アドレスマップが実装されている場合、サポートされているメモリ領域数は、次のうち大きい方です。

- サポートされているデータメモリ領域数
- サポートされている命令メモリ領域数

RGNR へのアクセス

RGNR にアクセスするには、`<opc1>` を 0、`<CRn>` を c6、`<CRm>` を c2、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c6,c2,0    ; Read  CP15 MPU Region Number Register
MCR p15,0,<Rt>,c6,c2,0    ; Write CP15 MPU Region Number Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.25 CP15 c7, Cache maintenance and other functions

The CP15 c7 registers are used for cache maintenance operations, and also provide barrier operations. Figure B4-8 shows the CP15 c7 registers.

Figure B4-8 CP15 c7 registers in a PMSA implementation

All CP15 c7 encodings not shown in Figure B4-8 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B4-27.

The CP15 c7 operations are described in:

- *CP15 c7, Cache and branch predictor maintenance functions*
- *CP15 c7, Miscellaneous functions* on page B4-72.

B4.6.26 CP15 c7, Cache and branch predictor maintenance functions

CP15 c7 provides a number of functions. This section describes only the CP15 c7 cache and branch predictor maintenance operations. Branch predictor operations are included in this section, because they operate in a similar way to the cache maintenance operations.

Note

ARMv7 introduces significant changes in the CP15 c7 operations. Most of these changes are because, from ARMv7, the architecture covers multiple levels of cache. This section only describes the ARMv7 requirements for these operations. For details of these operations in previous versions of the architecture see:

- *c7, Cache operations* on page AppxG-38 for ARMv6
- *c7, Cache operations* on page AppxH-49 for ARMv4 and ARMv5.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.25 CP15 c7、キャッシュ保守機能とその他の機能

CP15 c7 レジスタは、キャッシュ保守操作に使用されると同時に、バリア操作も提供します。CP15 c7 レジスタを図 B4-8 に示します。

図 B4-8 PMSA 実装の CP15 c7 レジスタ

図 B4-8 に示されていない CP15 c7 レジスタのすべてのエンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

次のセクションで、CP15 c7 操作について説明します。

- *CP15 c7、キャッシュおよび分岐予測器保守機能*
- *CP15 c7、その他の機能* P. B4-72

B4.6.26 CP15 c7、キャッシュおよび分岐予測器保守機能

CP15 c7 では複数の機能が提供されます。このセクションでは、CP15 c7 のキャッシュおよび分岐予測器の保守操作についてのみ説明します。分岐予測器保守操作については、キャッシュ保守操作と似ているため、このセクションで説明します。

注

ARMv7 では、CP15 c7 操作が大幅に変更されました。これらの変更のほとんどは、ARMv7 からアーキテクチャで複数のキャッシュレベルがカバーされたことによります。このセクションでは、これらの操作に関する ARMv7 の要件についてのみ説明します。以前のバージョンの ARM アーキテクチャにおけるこれらの操作の詳細については、次のセクションを参照して下さい。

- *c7、キャッシュ操作* P. 付録 G-38（ARMv6 の場合）
- *c7、キャッシュ操作* P. 付録 H-49 （ARMv4 と ARMv5 の場合）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure B4-9 shows the CP15 c7 cache and branch predictor maintenance operations.

Figure B4-9 CP15 c7 Cache and branch predictor maintenance operations

The CP15 c7 cache and branch predictor maintenance operations are all write-only operations that can be executed only in privileged modes. They are listed in Table B4-19.

For more information about the terms used in this section see *Terms used in describing cache operations* on page B2-10. The Multiprocessing Extensions changes the set of caches affected by these operations, see *Multiprocessor effects on cache maintenance operations* on page B2-23.

In Table B4-19, the *Rt data* column specifies what data is required in the register Rt specified by the `MCR` instruction used to perform the operation. For more information about the possible data formats see *Data formats for the cache and branch predictor operations* on page B4-70.

Table B4-19 CP15 c7 cache and branch predictor maintenance operations

CRm	opc2	Mnemonic	Function[a]	Rt data
c1	0	ICIALLUIS	Invalidate all instruction caches to PoU Inner Shareable. Also flushes branch target cache.[b]	Ignored
c1	1	BPIALLIS	Invalidate entire branch predictor array Inner Shareable.	Ignored
c5	0	ICIALLU	Invalidate all instruction caches to PoU. Also flushes branch target cache.[c]	Ignored
c5	1	ICIMVAU	Invalidate instruction cache line by address to PoU.[b, d]	Address
c5	6	BPIALL	Invalidate entire branch predictor array.	Ignored

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP15 c7 のキャッシュおよび分岐予測器の保守操作を、図 B4-9 に示します。

図 B4-9 CP15 c7 のキャッシュおよび分岐予測器の保守操作

CP15 c7 のキャッシュおよび分岐予測器の保守操作はすべて、特権モードでのみ実行可能な書き込み専用操作です。これらの操作の一覧を、表 B4-19 に示します。

このセクションで使用される用語の詳細については、*キャッシュ操作の説明で使用される用語* P. B2-10 を参照して下さい。マルチプロセッシング拡張機能が実装されている場合、これらの操作の影響を受けるキャッシュのセットが異なります。*キャッシュの保守操作に対するマルチプロセッサの影響* P. B2-23 を参照して下さい。

表 B4-19 で、*Rt データ*列には、操作を実行するために使用される `MCR` 命令で指定されるレジスタ Rt に、どのデータが必要かが示されています。使用可能なデータフォーマットの詳細については、*キャッシュおよび分岐予測器の操作のデータフォーマット* P. B4-70 を参照して下さい。

表 B4-19 CP15 c7 のキャッシュおよび分岐予測器の保守操作

CRm	opc2	ニーモニック	機能[a]	Rt データ
c1	0	ICIALLUIS	PoU まで、内部共有可能命令キャッシュ全体を無効化します。分岐先キャッシュのフラッシュも行います。[b]	無視
c1	1	BPIALLIS	内部共有可能分岐予測器アレイ全体を無効化します。	無視
c5	0	ICIALLU	PoU まで、すべての命令キャッシュを無効化します。分岐先キャッシュのフラッシュも行います。[c]	無視
c5	1	ICIMVAU	PoU まで、アドレスによって命令キャッシュラインを無効化します。[b,d]	アドレス
c5	6	BPIALL	分岐予測器アレイ全体を無効化します。	無視
c5	7	BPIMVA	内部共有可能ドメインの分岐予測器アレイからアドレスを無効化します。[d]	アドレス

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B4-19 CP15 c7 cache and branch predictor maintenance operations (continued)

CRm	opc2	Mnemonic	Function[a]	Rt data
c5	7	BPIMVA	Invalidate address from branch predictor array in the inner shareable domain.[d]	Address
c6	1	DCIMVAC	Invalidate data or unified cache line by address to PoU.[d]	Address
c6	2	DCISW	Invalidate data or unified cache line by set/way.	Set/way
c10	1	DCCMVAC	Clean data or unified cache line by address to PoC.[d]	Address
c10	2	DCCSW	Clean data or unified cache line by set/way.	Set/way
c11	1	DCCMVAU	Clean data or unified cache line by address to PoU.[d]	Address
c14	1	DCCIMVAC	Clean and invalidate data or unified cache line by address to PoC.[d]	Address
c14	2	DCCISW	Clean and invalidate data or unified cache line by set/way.	Set/way

a. Address, *point of coherency* (PoC) and *point of unification* (PoU) are described in *Terms used in describing cache operations* on page B2-10.
b. Only applies to separate instruction caches, does not apply to unified caches.
c. Only applies to separate instruction caches, does not apply to unified caches.
d. In general descriptions of the cache operations, these functions are described as operating by MVA (Modified Virtual Address). In a PMSA implementation the MVA and the PA have the same value, and so the functions operate using a physical address in the memory map.

Data formats for the cache and branch predictor operations

Table B4-19 on page B4-69 shows three possibilities for the data in the register Rt specified by the `MCR` instruction. These are described in the following subsections:

- *Ignored*
- *Address*
- *Set/way* on page B4-71

Ignored

The value in the register specified by the `MCR` instruction is ignored. You do not have to write a value to the register before issuing the `MCR` instruction.

Address

In general descriptions of the maintenance operations, operations that require a memory address are described as operating *by MVA*. For more information, see *Terms used in describing cache operations* on page B2-10. In a PMSA implementation, these operations require the physical address in the memory map. When the data is stated to be an address, it does not have to be cache line aligned.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 B4-19 CP15 c7 のキャッシュおよび分岐予測器の保守操作（続き）

CRm	opc2	ニーモニック	機能[a]	Rt データ
c6	1	DCIMVAC	PoU まで、アドレスによってデータまたは統一キャッシュラインを無効化します。[d]	アドレス
c6	2	DCISW	セット / ウェイによって、データまたは統一キャッシュラインを無効化します。	セット / ウェイ
c10	1	DCCMVAC	PoC まで、アドレスによってデータまたは統一キャッシュラインをクリーニングします。[d]	アドレス
c10	2	DCCSW	セット / ウェイによって、データまたは統一キャッシュラインをクリーニングします。	セット / ウェイ
c11	1	DCCMVAU	PoU まで、アドレスによってデータまたは統一キャッシュラインをクリーニングします。[d]	アドレス
c14	1	DCCIMVAC	PoC まで、アドレスによってデータまたは統一キャッシュラインをクリーニングおよび無効化します。[d]	アドレス
c14	2	DCCISW	セット / ウェイによって、データまたは統一キャッシュラインをクリーニングおよび無効化します。	セット / ウェイ

a. アドレス、*コヒーレンシ時点* (PoC)、*統一時点* (PoU) については、*キャッシュ操作の説明で使用される用語* P. B2-10 で説明します。
b. 独立した命令キャッシュにのみ適用されます。統一キャッシュには適用されません。
c. 独立した命令キャッシュにのみ適用されます。統一キャッシュには適用されません。
d. キャッシュ操作の一般的な説明では、これらの機能が MVA（修飾仮想アドレス）による操作として記述されます。PMSA 実装では MVA と PA が同じ値なため、これらの機能はメモリマップの物理アドレスを使用して動作します。

キャッシュおよび分岐予測器の操作のデータフォーマット

`MCR` 命令によって指定されるレジスタ R t に存在する可能性のある 3 種類のデータを、P. B4-69 の表 B4-19 に示します。これらのデータについては、次のサブセクションで説明します。

- *無視*
- *アドレス*
- *セット / ウェイ* P. B4-71

無視

`MCR` 命令で指定されたレジスタの値は無視されます。`MCR` 命令の発行前にレジスタに値を書き込む必要はありません。

アドレス

保守操作の一般的な説明では、メモリアドレスを必要とする操作は、*MVA による*操作として記述されます。詳細については、*キャッシュ操作の説明で使用される用語* P. B2-10 を参照して下さい。PMSA 実装では、これらの操作にメモリマップの物理アドレスが必要です。データがアドレスであると記載されている場合、キャッシュラインにアラインする必要はありません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Set/way

For an operation by set/way, the data identifies the cache line that the operation is to be applied to by specifying:

- the cache set the line belongs to
- the way number of the line in the set
- the cache level.

The format of the register data for a set/way operation is:

31 – 32-A	31-A – B	B-1 – L	L-1 – 4	3 – 1	0
Way	SBZ	Set	SBZ	Level	0

Where:

A = Log_2(ASSOCIATIVITY)

B = (L + S)

L = Log_2(LINELEN)

S = Log_2(NSETS)

ASSOCIATIVITY, LINELEN (Line Length) and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on.

The values of A and S are rounded up to the next integer.

Level ((Cache level to operate on) -1)

For example, this field is 0 for operations on L 1 cache, or 1 for operations on L 2 cache.

Set The number of the set to operate on.

Way The number of the way to operate on.

Note

- If L = 4 then there is no SBZ field between the set and level fields in the register.
- If A = 0 there is no way field in the register, and register bits [31:B] are SBZ.
- If the level, set or way field in the register is larger than the size implemented in the cache then the effect of the operation is UNPREDICTABLE.

Accessing the CP15 c7 cache maintenance operations

To perform one of the cache maintenance operations you write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c7, and `<CRm>` and `<opc2>` set to the values shown in Table B4-19 on page B4-69.

That is:

```
MCR p15,0,<Rt>,c7,<CRm>,<opc2>
```

For example:

```
MCR p15,0,<Rt>,c7,c5,0    ; Invalidate all instruction caches to point of unification
MCR p15,0,<Rt>,c7,c10,2   ; Clean data or unified cache line by set/way
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

セット／ウェイ

セット / ウェイによる操作の場合は、データで次の要素を指定することによって、操作が適用されるキャッシュラインが識別されます。

- ラインが属しているキャッシュセット
- セット内のラインのウェイ番号
- キャッシュレベル

セット / ウェイ操作のレジスタデータのフォーマットは次のとおりです。

31 – 32 – A	31 – A – B	B – 1 – L	L – 1 – 4	3 – 1	0
ウェイ	SBZ	セット	SBZ	レベル	0

各項目の説明については次を参照して下さい。

A = Log_2 (アソシエティビティ)

B = (L + S)

L = Log_2 (LINELEN)

S = Log_2 (NSETS)

ASSOCIATIVITY（アソシエティビティ）、LINELEN（ライン長）、NSETS（セット数）は、他の場所で使用されているのと同じ意味で、操作対象のキャッシュレベルに関する値です。

A と S の値は、それよりも大きい次の整数に丸められます。

レベル ((操作対象のキャッシュレベル) − 1)

例えば、このフィールドは、L1 キャッシュの操作では 0、L2 キャッシュの操作では 1 です。

セット 操作対象のセット番号

ウェイ 操作対象のウェイ番号

注

- L = 4 の場合、レジスタのセットフィールドとレベルフィールドとの間に SBZ フィールドは存在しません。
- A = 0 の場合は、レジスタにウェイフィールドが存在せず、レジスタのビット [31:B] は SBZ です。
- レジスタのレベル、セット、またはウェイフィールドが、キャッシュに実装されているサイズより大きい場合、操作の影響は予測不能です。

CP15 c7 キャッシュ保守操作へのアクセス

キャッシュ保守操作のいずれかを実行するには、`<opc1>` を 0、`<CRn>` を c7、`<CRm>` と `<opc2>` を P. B4- 69 の表 B4- 19 に示す値に設定して、CP15 レジスタに書き込みます。

次の形式になります。

```
MCR p15,0,<Rt>,c7,<CRm>,<opc2>
```

次に例を示します。

```
MCR p15,0,<Rt>,c7,c5,0    ; Invalidate all instruction caches to point of unification
MCR p15,0,<Rt>,c7,c10,2   ; Clean data or unified cache line by set/way
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.27 CP15 c7, Miscellaneous functions

CP15 c7 provides a number of functions, summarized in Figure B4-8 on page B4-68. This section describes only the CP15 c7 miscellaneous operations.

Figure B4-10 shows the CP15 c7 miscellaneous operations. It does not show the other CP15 c7 operations.

Figure B4-10 CP15 c7 Miscellaneous operations

The CP15 c7 miscellaneous operations are described in:

- *CP15 c7, Data and Instruction Barrier operations*
- *CP15 c7, No Operation (NOP)* on page B4-73.

CP15 c7, Data and Instruction Barrier operations

ARMv6 includes two CP15 c7 operations to perform Data Barrier operations, and another operation to perform an Instruction Barrier operation. In ARMv7:

- The ARM and Thumb instruction sets include instructions to perform the barrier operations, that can be executed in unprivileged and privileged modes, see *Memory barriers* on page A3-47.
- The CP15 c7 operations are defined as write-only operations, that can be executed in unprivileged and privileged modes, but using these operations is deprecated. The three operations are described in:
  - *Instruction Synchronization Barrier operation* on page B4-73
  - *Data Synchronization Barrier operation* on page B4-73
  - *Data Memory Barrier operation* on page B4-73.

  The value in the register Rt specified by the `MCR` instruction used to perform one of these operations is ignored. You do not have to write a value to the register before issuing the `MCR` instruction.

  In ARMv7 using these CP15 c7 operations is deprecated. Use the `ISB`, `DSB`, and `DMB` instructions instead.

Note

- In ARMv6 and earlier documentation, the Instruction Synchronization Barrier operation is referred to as a Prefetch Flush.
- In versions of the ARM architecture before ARMv6 the Data Synchronization Barrier operation is described as a *Data Write Barrier* (DWB).

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.27 CP15 c7、その他の機能

CP15 c7 では、P. B4-68 の図 B4-8 に示す複数の機能が提供されます。このセクションでは、CP15 c7 のその他の操作についてのみ説明します。

CP15 c7 のその他の操作を、図 B4-10 に示します。他の CP15 c7 操作は、この表には示されていません。

図 B4-10 CP15 c7、その他の操作

次のセクションで、CP15 c7 のその他の操作について説明します。

- *CP15 c7、データおよび命令バリア操作*
- *CP15 c7、無操作 (NOP)* P. B4-73

CP15 c7、データおよび命令バリア操作

ARMv6 には、データバリア操作を実行するための 2 つの CP15 c7 操作と、命令バリア操作を実行するための別の操作が含まれています。ARMv7 では、次の規則が適用されます。

- ARM 命令セットと Thumb 命令セットには、非特権モードと特権モードで実行可能なバリア操作を実行するための命令が含まれています（*メモリバリア* P. A3-47 を参照）。
- CP15 c7 操作は、非特権モードと特権モードで実行可能な書き込み専用操作として定義されますが、これらの操作の使用は非推奨です。これらの 3 つの操作については、次のサブセクションで説明します。
  - *命令同期バリア操作* P. B4-73
  - *データ同期バリア操作* P. B4-73
  - *データ メモリバリア操作* P. B4-73

  これらのいずれかの操作を実行するための `MCR` 命令で指定されたレジスタ Rt の値は無視されます。`MCR` 命令の発行前にレジスタに値を書き込む必要はありません。

  ARMv7 では、これらの CP15 c7 操作の使用は非推奨です。代わりに、`ISB`、`DSB`、`DMB` 命令を使用して下さい。

注

- ARMv6 およびそれ以前の説明書では、命令同期バリア操作がプリフェッチフラッシュと呼ばれていました。
- ARMv6 以前のバージョンの ARM アーキテクチャでは、データ同期バリア操作がデータ*書き込みバリア*(DWB) と呼ばれています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Instruction Synchronization Barrier operation

In ARMv7, the ISB instruction is used to perform an Instruction Synchronization Barrier, see *ISB* on page A8-102.

The deprecated CP15 c7 encoding for an Instruction Synchronization Barrier is <opc1> set to 0, <CRn> set to c7, <CRm> set to c5, and <opc2> set to 4.

Data Synchronization Barrier operation

In ARMv7, the DSB instruction is used to perform a Data Synchronization Barrier, see *DSB* on page A8-92.

The deprecated CP15 c7 encoding for a Data Synchronization Barrier is <opc1> set to 0, <CRn> set to c7, <CRm> set to c10, and <opc2> set to 4. This operation performs the full system barrier performed by the DSB instruction.

Data Memory Barrier operation

In ARMv7, the DMB instruction is used to perform a Data Memory Barrier, see *DMB* on page A8-90.

The deprecated CP15 c7 encoding for a Data Memory Barrier is <opc1> set to 0, <CRn> set to c7, <CRm> set to c10, and <opc2> set to 5. This operation performs the full system barrier performed by the DMB instruction.

CP15 c7, No Operation (NOP)

ARMv6 includes two CP15 c7 operations that are not supported in ARMv7, with encodings that become *No Operation* (NOP) in ARMv7. These are:

- The *Wait For Interrupt* (CP15WFI) operation. In ARMv7 this operation is performed by the WFI instruction, that is available in the ARM and Thumb instruction sets. For more information, see *WFI* on page A8-810.
- The prefetch instruction by MVA operation. In ARMv7 this operation is replaced by the PLI instruction, that is available in the ARM and Thumb instruction sets. For more information, see *PLI (immediate, literal)* on page A8-242, and *PLI (register)* on page A8-244.

In ARMv7, the CP15 c7 encodings that were used for these operations must be valid write-only operations that perform a NOP. These encodings are:

- for the ARMv6 CP15WFI operation:
  - <opc1> set to 0, <CRn> set to c7, <CRm> set to c0, and <opc2> set to 4
- for the ARMv6 prefetch instruction by MVA operation:
  - <opc1> set to 0, <CRn> set to c7, <CRm> set to c13, and <opc2> set to 1.

B4.6.28 CP15 c8, Not used on a PMSA implementation

CP15 c8 is not used on an ARMv7-R implementation, see *Unallocated CP15 encodings* on page B4-27.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

命令同期バリア操作

ARMv7 では、命令同期バリアを実行するために ISB 命令が使用されます。*ISB* P. A8-102 を参照して下さい。

命令同期バリア用の CP15 c7 エンコードは、<opc1> を 0、<CRn> を c7、<CRm> を c5、<opc2> を 4 に設定しますが、このエンコードは非推奨です。

データ同期バリア操作

ARMv7 では、データ同期バリアを実行するために DSB 命令が使用されます。*DSB* P. A8-92 を参照して下さい。

データ同期バリア用の CP15 c7 エンコードは、<opc1> を 0、<CRn> を c7、<CRm> を c10、<opc2> を 4 に設定しますが、このエンコードは非推奨です。この操作では、DSB 命令で実行される完全なシステムバリアが実行されます。

データ メモリバリア操作

ARMv7 では、データ メモリバリアを実行するために DMB 命令が使用されます。*DMB* P. A8-90 を参照して下さい。

データ メモリバリア用の CP15 c7 エンコードは、<opc1> を 0、<CRn> を c7、<CRm> を c10、<opc2> を 5 に設定しますが、このエンコードは非推奨です。この操作では、DMB 命令で実行される完全なシステムバリアが実行されます。

CP15 c7、無操作 (NOP)

ARMv6 には、ARMv7 では*無操作*(NOP) になるエンコードを使用した、ARMv7 ではサポートされていない 2 つの CP15 c7 操作が含まれています。具体的には、次のものがあります。

- *割り込み待ち*(CP15WFI) 操作。ARMv7 では、この操作が、ARM および Thumb 命令セットで使用可能な WFI 命令によって実行されます。詳細については、*WFI* P. A8-810 を参照して下さい。
- MVA による命令のプリフェッチ操作。ARMv7 では、この操作が、ARM および Thumb 命令セットで使用可能な PLI 命令に置き換えられます。詳細については、*PLI（イミディエート、リテラル）* P. A8-242 と *PLI（レジスタ）* P. A8-244 を参照して下さい。

ARMv7 では、これらの操作用に使用されていた CP15 c7 エンコードを、NOP を実行する有効な書き込み専用操作にする必要があります。これらのエンコードは次のとおりです。

- ARMv6 CP15WFI 操作の場合
  - <opc1> を 0、<CRn> を c7、<CRm> を c0、<opc2> を 4 に設定
- ARMv6 MVA による命令のプリフェッチ操作の場合
  - <opc1> を 0、<CRn> を c7、<CRm> を c13、<opc2> を 1 に設定

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.29 CP15 c9, Cache and TCM lockdown registers and performance monitors

Some CP15 c9 register encodings are reserved for IMPLEMENTATION DEFINED memory system functions, in particular:

- cache control, including lockdown
- TCM control, including lockdown
- branch predictor control.

Additional CP15 c9 encodings are reserved for performance monitors. These encodings fall into two groups:

- the optional performance monitors described in Chapter C9 *Performance Monitors*
- additional IMPLEMENTATION DEFINED performance monitors.

The reserved encodings permit implementations that are compatible with previous versions of the ARM architecture, in particular with the ARMv6 requirements. Figure B4-11 shows the permitted CP15 c9 register encodings.

Figure B4-11 Permitted CP15 c9 register encodings

All CP15 c9 encodings not shown in Figure B4-11 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B4-27.

In ARMv6, CP15 c9 provides cache lockdown functions. With the ARMv7 abstraction of the hierarchical memory model, for CP15 c9:

- All encodings with CRm = {c0-c2, c5-c8} are reserved for IMPLEMENTATION DEFINED cache, branch predictor and TCM operations.

  This reservation enables the implementation of a scheme that is backwards compatible with ARMv6. For details of the ARMv6 implementation see *c9, Cache lockdown support* on page AppxG-45.

- All encodings with CRm = {c12-c14} are reserved for the optional performance monitors that are defined in Chapter C9 *Performance Monitors*.

- All encodings with CRm = c15 are reserved for IMPLEMENTATION DEFINED performance monitoring features.

B4.6.30 CP15 c10, Not used on a PMSA implementation

CP15 c10 is not used on an ARMv7-R implementation, see *Unallocated CP15 encodings* on page B4-27.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.28 CP15 c8、PMSA 実装では未使用

ARMv7-R 実装では CP15 c8 が使用されません。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

B4.6.29 CP15 c9、キャッシュおよび TCM ロックダウンレジスタとパフォーマンスモニタ

一部の CP15 c9 レジスタのエンコードが、実装定義のメモリシステム機能、特に次の機能のため予約されています。

- ロックダウンを含むキャッシュ制御
- ロックダウンを含む TCM 制御
- 分岐予測器の制御

追加の CP15 c9 エンコードが、パフォーマンスモニタ用として予約されています。これらのエンコードは、次の 2 つのグループに分類されます。

- 第 C9 章「*パフォーマンスモニタ*」で説明されているオプションのパフォーマンスモニタ
- 追加の実装定義のパフォーマンスモニタ

予約されているエンコードでは、特に ARMv6 要件を伴う、以前のバージョンの ARM アーキテクチャと互換性のある実装が許可されます。許可される CP15 c9 レジスタのエンコードを、図 B4-11 に示します。

図 B4-11 許可される CP15 c9 レジスタのエンコード

図 B4-11 に示されていないすべての CP15 c9 エンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

ARMv6 では、CP15 c9 によってキャッシュロックダウン機能が提供されます。階層型メモリモデルの ARMv7 抽象化を使用した CP15 c9 の場合、次の規則が適用されます。

- CRm == {c0-c2,c5-c8} を使用したすべてのエンコードは、実装定義のキャッシュ、分岐予測器、TCM 操作用として予約されています。

  この予約によって、ARMv6 と下位互換性のある方式の実装が可能になります。ARMv6 実装の詳細については、*c9、キャッシュロックダウンのサポート* P. 付録 G-45 を参照して下さい。

- CRm = {c12-c14} であるすべてのエンコードは、第 C9 章「*パフォーマンスモニタ*」で定義されているオプションのパフォーマンスモニタ用として予約されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.31 CP15 c11, Reserved for TCM DMA registers

Some CP15 c11 register encodings are reserved for IMPLEMENTATION DEFINED DMA operations to and from TCM, see Figure B4-12.

Figure B4-12 Permitted CP15 c11 register encodings

All CP15 c11 encodings not shown in Figure B4-12 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B4-27.

B4.6.32 CP15 c12, Not used on a PMSA implementation

CP15 c12 is not used on an ARMv7-R implementation, see *Unallocated CP15 encodings* on page B4-27.

B4.6.33 CP15 c13, Context and Thread ID registers

The CP15 c13 registers are used for:

- a Context ID register
- three software Thread ID registers.

Figure B4-13 shows the CP15 c13 registers:

Figure B4-13 CP15 c13 registers in a PMSA implementation

All CP15 c13 encodings not shown in Figure B4-13 are UNPREDICTABLE, see *Unallocated CP15 encodings* on page B4-27.

The CP15 c13 registers are described in:

- *c13, Context ID Register (CONTEXTIDR)* on page B4-76
- *CP15 c13 Software Thread ID registers* on page B4-77.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- CRm = c15 であるすべてのエンコードは、実装定義のパフォーマンスモニタ機能用として予約されています。

B4.6.30 CP15 c10、PMSA 実装では未使用

ARMv7-R 実装では CP15 c10 が使用されません。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

B4.6.31 CP15 c11、TCM DMA 用に予約されているレジスタ

一部の CP15 c11 レジスタのエンコードが、TCM に対する実装定義の DMA 操作用として予約されています。図 B4-12 を参照して下さい。

図 B4-12 許可される CP15 c11 レジスタのエンコード

図 B4-12 に示されていないすべての CP15 c11 エンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

B4.6.32 CP15 c12、PMSA 実装では未使用

ARMv7-R 実装では CP15 c12 が使用されません。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

B4.6.33 CP15 c13、コンテキストおよびスレッド ID レジスタ

CP15 c13 レジスタの用途は次のとおりです。

- コンテキスト ID レジスタ
- 3 つのソフトウェアスレッド ID レジスタ

CP15 c13 レジスタを、図 B4-13 に示します。

図 B4-13 PMSA 実装の CP15 c13 レジスタ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.34 c13, Context ID Register (CONTEXTIDR)

The Context ID Register, CONTEXTIDR, identifies the current context by means of a *Context Identifier* (Context ID).

Note

Previously, on PMSA implementations, this Context ID has been described as a *Process Identifier* (PROCID), and this CP15 c13 register has been called the *Process ID Register*. The new naming makes the register naming consistent for PMSA and VMSA implementations.

The whole of this register is used by:

- the debug logic, for Linked and Unlinked Context ID matching, see *Breakpoint debug events* on page C3-5 and *Watchpoint debug events* on page C3-15.
- the trace logic, to identify the current process.

The CONTEXTIDR is:

- a 32-bit read/write register
- accessible only in privileged modes.

The format of the CONTEXTIDR is:

31	0
ContextID	

ContextID, bits [31:0]

Context Identifier. This field must be programmed with a unique context identifier value that identifies the current process. It is used by the trace logic and the debug logic to identify the process that is running currently.

Accessing the CONTEXTIDR

To access the CONTEXTIDR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c13, `<CRm>` set to c0, and `<opc2>` set to 1. For example:

```
MRC p15,0,<Rt>,c13,c0,1    ; Read  CP15 Context ID Register
MCR p15,0,<Rt>,c13,c0,1    ; Write CP15 Context ID Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

図 B4-13 に示されていないすべての CP15 c13 エンコードは、予測不能です。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

CP15 c13 レジスタについては、次のセクションで説明します。

- *c13、コンテキスト ID レジスタ (CONTEXTIDR)* P. B4-76
- *CP15 c13、ソフトウェアスレッド ID レジスタ* P. B4-77

B4.6.34 c13、コンテキスト ID レジスタ (CONTEXTIDR)

コンテキスト ID レジスタ (CONTEXTIDR) では、*コンテキスト識別子*（コンテキスト ID）を使用して現在のコンテキストが識別されます。

注

以前の PMSA 実装では、このコンテキスト ID が*プロセス識別子*(PROCID) として表記され、この CP15 c13 レジスタが*プロセス ID レジスタ*と呼ばれていました。この新しい命名によって、PMSA 実装と VMSA 実装のレジスタ名の整合性が取れました。

このレジスタ全体は、次の用途に使用されます。

- リンク付きおよびリンクなしのコンテキスト ID 一致用のデバッグロジック（*ブレークポイントデバッグ イベント* P. C3-5 と *ウォッチポイントデバッグ イベント* P. C3-15 を参照）
- トレースロジックで、現在のプロセスを識別するため

CONTEXTIDR の特徴は次のとおりです。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能

CONTEXTIDR のフォーマットは次のとおりです。

31	0
ContextID	

ContextID、ビット [31:0]

コンテキスト識別子。このフィールドは、現在のプロセスを識別する、一意なコンテキスト識別子の値にプログラムする必要があります。この値は、トレースロジックとデバッグロジックで、実行中のプロセスを識別するために使用されます。

CONTEXTIDR へのアクセス

CONTEXTIDR にアクセスするには、`<opc1>` を 0、`<CRn>` を c13、`<CRm>` を c0、`<opc2>` を 1 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c13,c0,1    ; Read  CP15 Context ID Register
MCR p15,0,<Rt>,c13,c0,1    ; Write CP15 Context ID Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.35 CP15 c13 Software Thread ID registers

The Software Thread ID registers provide locations where software can store thread identifying information, for OS management purposes. These registers are never updated by the hardware.

The Software Thread ID registers are:

- three 32-bit register read/write registers:
  - User Read/Write Thread ID Register, TPIDRURW
  - User Read-only Thread ID Register, TPIDRURO
  - Privileged Only Thread ID Register, TPIDRPRW.
- accessible in different modes:
  - the User Read/Write Thread ID Register is read/write in unprivileged and privileged modes
  - the User Read-only Thread ID Register is read-only in User mode, and read/write in privileged modes
  - the Privileged Only Thread ID Register is only accessible in privileged modes, and is read/write
- introduced in ARMv7.

Accessing the Software Thread ID registers

To access the Software Thread ID registers you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c13, `<CRm>` set to c0, and `<opc2>` set to:

- 2 for the User Read/Write Thread ID Register, TPIDRURW
- 3 for the User Read-only Thread ID Register, TPIDRURO
- 4 for the Privileged Only Thread ID Register, TPIDRPRW.

For example:

```
MRC p15, 0, <Rt>, c13, c0, 2 ; Read  CP15 User Read/Write Thread ID Register
MCR p15, 0, <Rt>, c13, c0, 2 ; Write CP15 User Read/Write Thread ID Register
MRC p15, 0, <Rt>, c13, c0, 3 ; Read  CP15 User Read-only  Thread ID Register
MCR p15, 0, <Rt>, c13, c0, 3 ; Write CP15 User Read-only  Thread ID Register
MRC p15, 0, <Rt>, c13, c0, 4 ; Read  CP15 Privileged Only Thread ID Register
MCR p15, 0, <Rt>, c13, c0, 4 ; Write CP15 Privileged Only Thread ID Register
```

B4.6.36 CP15 c14, Not used

CP15 c14 is not used on any ARMv7 implementation, see *Unallocated CP15 encodings* on page B4-27.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.35 CP15 c13、ソフトウェアスレッド ID レジスタ

ソフトウェアスレッド ID レジスタは、OS で管理するためのスレッド識別情報を、ソフトウェアで保存するための場所を提供します。これらのレジスタがハードウェアで更新されることはありません。

ソフトウェアスレッド ID レジスタの特徴は次のとおりです。

- 3 つの 32 ビットの読み出し / 書き込みレジスタ
  - ユーザ読み出し / 書き込みスレッド ID レジスタ、TPIDRURW
  - ユーザ読み出し専用スレッド ID レジスタ、TPIDRURO
  - 特権専用スレッド ID レジスタ、TPIDRPRW
- さまざまなモードでアクセス可能です。
  - ユーザ読み出し / 書き込みスレッド ID レジスタは、非特権モードと特権モードで読み出し / 書き込み可能です。
  - ユーザ読み出し専用スレッド ID レジスタは、ユーザモードで読み取り専用、特権モードで読み出し / 書き込み可能です。
  - 特権専用スレッド ID レジスタは、特権モードでのみアクセス可能で、読み出し / 書き込み可能です。
- ARMv7 で導入されました。

ソフトウェアスレッド ID レジスタへのアクセス

ソフトウェアスレッド ID レジスタにアクセスするには、`<opc1>` を 0、`<CRn>` を c13、`<CRm>` を c0、`<opc2>` を次の値に設定して、CP15 レジスタを読み出し / 書き込みします。

- ユーザ読み出し / 書き込みスレッド ID レジスタ (TPIDRURW) の場合は 2
- ユーザ読み出し専用スレッド ID レジスタ (TPIDRURO) の場合は 3
- 特権専用スレッド ID レジスタ (TPIDRPRW) の場合は 4

次に例を示します。

```
MRC p15, 0, <Rt>, c13, c0, 2 ; Read  CP15 User Read/Write Thread ID Register
MCR p15, 0, <Rt>, c13, c0, 2 ; Write CP15 User Read/Write Thread ID Register
MRC p15, 0, <Rt>, c13, c0, 3 ; Read  CP15 User Read-only  Thread ID Register
MCR p15, 0, <Rt>, c13, c0, 3 ; Write CP15 User Read-only  Thread ID Register
MRC p15, 0, <Rt>, c13, c0, 4 ; Read  CP15 Privileged Only Thread ID Register
MCR p15, 0, <Rt>, c13, c0, 4 ; Write CP15 Privileged Only Thread ID Register
```

B4.6.36 CP15 c4、未使用

ARMv7 実装では CP15 c14 が使用されません。*未割り当ての CP15 エンコード* P. B4-27 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.6.37 CP15 c15, IMPLEMENTATION DEFINED registers

CP15 c15 is reserved for IMPLEMENTATION DEFINED purposes. ARMv7 does not impose any restrictions on the use of the CP15 c15 encodings. The documentation of the ARMv7 implementation must describe fully any registers implemented in CP15 c15. Normally, for processor implementations by ARM, this information is included in the *Technical Reference Manual* for the processor.

Typically, CP15 c15 is used to provide test features, and any required configuration options that are not covered by this manual.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B4.6.37 CP15 c15、実装定義のレジスタ

CP15 c15 は実装定義の目的に予約されています。ARMv7 では、CP15 c15 エンコードの使用に関する制限はありません。ARMv7 実装の説明書には、CP15 c15 に実装されるすべてのレジスタを完全に記載する必要があります。通常、ARM によるプロセッサ実装の場合は、この情報がプロセッサの*テクニカルリファレンス マニュアル*に含まれています。

多くの場合、CP15 c15 は、テスト機能と、本書でカバーされていない必要な構成オプションを提供するために使用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.7 Pseudocode details of PMSA memory system operations

This section contains pseudocode describing PMSA-specific memory operations. The following subsections describe the pseudocode functions:

- *Alignment fault*
- *Address translation*
- *Default memory map attributes* on page B4-81.

See also the pseudocode for general memory system operations in *Pseudocode details of general memory system operations* on page B2-29.

B4.7.1 Alignment fault

The following pseudocode describes the Alignment fault in a PMSA implementation:

```
// AlignmentFaultP()
// =================

AlignmentFaultP(bits(32) address, boolean iswrite)

    DataAbort(address, bits(4) UNKNOWN, boolean UNKNOWN, iswrite, DAbort_Alignment);
```

B4.7.2 Address translation

The following pseudocode describes address translation in a PMSA implementation:

```
// TranslateAddressP()
// ===================

AddressDescriptor TranslateAddressP(bits(32) va, boolean ispriv, boolean iswrite)

    AddressDescriptor result;
    Permissions perms;

    // PMSA only does flat mapping and security domain is effectively IMPLEMENTATION DEFINED.
    result.paddress.physicaladdress = va;
    result.paddress.physicaladdressext = '00000000';
    IMPLEMENTATION_DEFINED setting of result.paddress.NS;

    if SCTLR.M == 0 then  // MPU is disabled

        result.memAttrs = DefaultMemoryAttributes(va);

    else                  // MPU is enabled

        // Scan through regions looking for matching ones. If found, the last
        // one matched is used.
        region_found = FALSE;

        for r=0 to MPUIR.DRegion-1
            size_enable    = DRSR[r];
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.7 PMSA メモリシステム操作の擬似コードの詳細

このセクションでは、PMSA 固有のメモリ操作を示す擬似コードを紹介します。次のサブセクションで、擬似コード関数について説明します。

- *アライメントフォールト*
- *アドレス変換*
- *デフォルトのメモリマップ属性* P. B4-81

一般的なメモリシステム操作の擬似コードの詳細 P. B2-29 にある、一般的なメモリシステム操作の擬似コードも参照して下さい。

B4.7.1 アライメントフォールト

次の擬似コードは、PMSA 実装のアライメントフォールトを示しています。

```
// AlignmentFaultP()
// =================

AlignmentFaultP(bits(32) address, boolean iswrite)

    DataAbort(address, bits(4) UNKNOWN, boolean UNKNOWN, iswrite, DAbort_Alignment);
```

B4.7.2 アドレス変換

次の擬似コードは、PMSA 実装のアドレス変換を示しています。

```
// TranslateAddressP()
// ===================

AddressDescriptor TranslateAddressP(bits(32) va, boolean ispriv, boolean iswrite)

    AddressDescriptor result;
    Permissions perms;

    // PMSA only does flat mapping and security domain is effectively IMPLEMENTATION DEFINED.
    result.paddress.physicaladdress = va;
    result.paddress.physicaladdressext = '00000000';
    IMPLEMENTATION_DEFINED setting of result.paddress.NS;

    if SCTLR.M == 0 then  // MPU is disabled

        result.memAttrs = DefaultMemoryAttributes(va);

    else                  // MPU is enabled

        // Scan through regions looking for matching ones. If found, the last
        // one matched is used.
        region_found = FALSE;

        for r=0 to MPUIR.DRegion-1
            size_enable    = DRSR[r];
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
            base_address   = DRBAR[r];
            access_control = DRACR[r];

            if size_enable<0> == '1' then  // Region is enabled
                lsbit = UInt(size_enable<5:1>) + 1;
                if lsbit < 2 then UNPREDICTABLE;

                if lsbit == 32 || va<31:lsbit> == base_address<31:lsbit> then
                    if lsbit >= 8 then  // can have subregions
                        subregion = UInt(va<lsbit-1:lsbit-3>);
                        hit = (size_enable<subregion+8> == '0');
                    else
                        hit = TRUE;

                    if hit then
                        texcb = access_control<5:3,1:0>;
                        S = access_control<2>;
                        perms.ap = access_control<10:8>;
                        perms.xn = access_control<12>;
                        region_found = TRUE;

        // Generate the memory attributes, and also the permissions if no region found.
        if region_found then
            result.mematrs = DefaultTEXDecode(texcb, S);
        else
            if SCTLR.BR == '0' || NOT(ispriv) then
                DataAbort(address, bits(4) UNKNOWN, boolean UNKNOWN, iswrite, DAbort_Background);
            else
                result.mematrs = DefaultMemoryAttributes(va);
                perms.ap = '011';
                perms.xn = if va<31:28> == '1111' then NOT(SCTLR.V) else va<31>;

        // Check the permissions.
        CheckPermission(perms, VA, boolean UNKNOWN, bits(4) UNKNOWN, iswrite, ispriv);

    return result;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
            base_address   = DRBAR[r];
            access_control = DRACR[r];

            if size_enable<0> == '1' then  // Region is enabled
                lsbit = UInt(size_enable<5:1>) + 1;
                if lsbit < 2 then UNPREDICTABLE;

                if lsbit == 32 || va<31:lsbit> == base_address<31:lsbit> then
                    if lsbit >= 8 then  // can have subregions
                        subregion = UInt(va<lsbit-1:lsbit-3>);
                        hit = (size_enable<subregion+8> == '0');
                    else
                        hit = TRUE;

                    if hit then
                        texcb = access_control<5:3,1:0>;
                        S = access_control<2>;
                        perms.ap = access_control<10:8>;
                        perms.xn = access_control<12>;
                        region_found = TRUE;

        // Generate the memory attributes, and also the permissions if no region found.
        if region_found then
            result.mematrs = DefaultTEXDecode(texcb, S);
        else
            if SCTLR.BR == '0' || NOT(ispriv) then
                DataAbort(address, bits(4) UNKNOWN, boolean UNKNOWN, iswrite, DAbort_Background);
            else
                result.mematrs = DefaultMemoryAttributes(va);
                perms.ap = '011';
                perms.xn = if va<31:28> == '1111' then NOT(SCTLR.V) else va<31>;

        // Check the permissions.
        CheckPermission(perms, VA, boolean UNKNOWN, bits(4) UNKNOWN, iswrite, ispriv);

    return result;
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.7.3 Default memory map attributes

The following pseudocode describes the default memory map attributes in a PMSA implementation:

```
// DefaultMemoryAttributes()
// =========================

MemoryAttributes DefaultMemoryAttributes(bits(32) va)

    MemoryAttributes memattrs;

    case va<31:30> of
        when '00'
            if SCTLR.C == '0' then
                memattrs.type = MemType_Normal;
                memattrs.innerattrs = '00';  // Non-cacheable
                memattrs.shareable = TRUE;
            else
                memattrs.type = MemType_Normal;
                memattrs.innerattrs = '01';  // Write-back write-allocate cacheable
                memattrs.shareable = FALSE;
        when '01'
            if SCTLR.C == '0' || va<29> == '1' then
                memattrs.type = MemType_Normal;
                memattrs.innerattrs = '00';  // Non-cacheable
                memattrs.shareable = TRUE;
            else
                memattrs.type = MemType_Normal;
                memattrs.innerattrs = '10';  // Write-through cacheable
                memattrs.shareable = FALSE;
        when '10'
            memattrs.type = MemType_Device;
            memattrs.innerattrs = '00';      // Non-cacheable
            memattrs.shareable = (va<29> == '1');
        when '11'
            memattrs.type = MemType_StronglyOrdered;
            memattrs.innerattrs = '00';      // Non-cacheable
            memattrs.shareable = TRUE;

        // Outer attributes are the same as the inner attributes in all cases.
        memattrs.outerattrs = memattrs.innerattrs;
        memattrs.outershareable = memattrs.shareable;

    return memattrs;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B4.7.3 デフォルトのメモリマップ属性

次の擬似コードは、PMSA 実装のデフォルトのメモリマップ属性を示しています。

```
// DefaultMemoryAttributes()
// =========================

MemoryAttributes DefaultMemoryAttributes(bits(32) va)

    MemoryAttributes memattrs;

    case va<31:30> of
        when '00'
            if SCTLR.C == '0' then
                memattrs.type = MemType_Normal;
                memattrs.innerattrs = '00';  // Non-cacheable
                memattrs.shareable = TRUE;
            else
                memattrs.type = MemType_Normal;
                memattrs.innerattrs = '01';  // Write-back write-allocate cacheable
                memattrs.shareable = FALSE;
        when '01'
            if SCTLR.C == '0' || va<29> == '1' then
                memattrs.type = MemType_Normal;
                memattrs.innerattrs = '00';  // Non-cacheable
                memattrs.shareable = TRUE;
            else
                memattrs.type = MemType_Normal;
                memattrs.innerattrs = '10';  // Write-through cacheable
                memattrs.shareable = FALSE;
        when '10'
            memattrs.type = MemType_Device;
            memattrs.innerattrs = '00';      // Non-cacheable
            memattrs.shareable = (va<29> == '1');
        when '11'
            memattrs.type = MemType_StronglyOrdered;
            memattrs.innerattrs = '00';      // Non-cacheable
            memattrs.shareable = TRUE;

        // Outer attributes are the same as the inner attributes in all cases.
        memattrs.outerattrs = memattrs.innerattrs;
        memattrs.outershareable = memattrs.shareable;

    return memattrs;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter B5
The CPUID Identification Scheme

This chapter describes the CPUID scheme introduced as a requirement in ARMv7. This scheme provides registers that identify the architecture version and many features of the processor implementation. This chapter also describes the registers that identify the implemented Advanced SIMD and VFP features, if any.

This chapter contains the following sections:

- *Introduction to the CPUID scheme* on page B5-2
- *The CPUID registers* on page B5-4
- *Advanced SIMD and VFP feature identification registers* on page B5-34.

Note

The other chapters of this manual describe the permitted combinations of architectural features for the ARMv7-A and ARMv7-R architecture profiles, and some of the appendices give this information for previous versions of the architecture. Typically, permitted features are associated with a named architecture version, or version and profile, such as ARMv7-A or ARMv6.

The CPUID scheme is a mechanism for describing these permitted combinations in a way that enables software to determine the capabilities of the hardware it is running on.

The CPUID scheme does not extend the permitted combinations of architectural features beyond those associated with named architecture versions and profiles. The fact that the CPUID scheme can describe other combinations does not imply that those combinations are permitted ARM architecture variants.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 B5 章
CPUID 識別方式

本章では、ARMv7 で要件の 1 つとして導入された CPUID 方式について説明します。この方式では、プロセッサ実装のアーキテクチャのバージョンと、さまざまな機能を識別するレジスタが提供されます。本章では、実装されているアドバンスト SIMD 機能と VFP 機能を識別するレジスタについても説明します。

本章は次のセクションから構成されています。

- *CPUID 方式の概要* P. B5-2
- *CPUID レジスタ* P. B5-4
- *アドバンスト SIMD および VFP 機能識別レジスタ* P. B5-34

注

本書の他の章では、ARMv7-A および ARMv7-R のアーキテクチャプロファイルで許可されるアーキテクチャ機能の組み合わせについて説明します。また付録の一部では、以前のバージョンのアーキテクチャに関する同様の情報を提供します。通常、許可された機能は、ARMv7-A や ARMv6 などの、名前付きのアーキテクチャのバーションか、バージョンとプロファイルに関連付けられます。

CPUID 方式は、ソフトウェアが実行されているハードウェアの機能を、ソフトウェアから判断できるように、これらの許可された組み合わせを記述するための機構です。

CPUID 方式は、許可されたアーキテクチャ機能の組み合わせを、名前付きのアーキテクチャのバージョンやプロファイルと関連付けされたもの以上に拡張するものではありません。CPUID 方式で他の組み合わせが記述可能である事実は、その組み合わせが ARM アーキテクチャバリアントとして許可されることを意味しません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B5.1 Introduction to the CPUID scheme

In ARM architecture versions before ARMv7, the architecture version is indicated by the Architecture field in the Main ID Register, see:

- *c0, Main ID Register (MIDR)* on page B3-81, for a VMSA implementation
- *c0, Main ID Register (MIDR)* on page B4-32, for a PMSA implementation.

From ARMv7, the architecture implements an extended processor identification scheme, using a number of registers in CP15 c0. ARMv7 requires the use of this scheme, and use of the scheme is indicated by a value of `0xF` in the Architecture field of the Main ID Register.

Note

Some ARMv6 processors implemented the scheme before its formal adoption in the architecture.

The CPUID scheme provides information about the implemented:

- processor features
- debug features
- auxiliary features, in particular IMPLEMENTATION DEFINED features
- memory model features
- instruction set features.

The following sections give more information about the CPUID registers:

- *Organization of the CPUID registers*
- *General features of the CPUID registers* on page B5-3.

The CPUID registers on page B5-4 gives detailed descriptions of the registers.

This chapter also describes the identification registers for any Advanced SIMD or VFP implementation. These are registers in the shared register space for the Advanced SIMD and VFP extensions, in CP 10 and CP 11. *Advanced SIMD and VFP feature identification registers* on page B5-34 describes these registers.

B5.1.1 Organization of the CPUID registers

Figure B5-1 on page B5-3 shows the CPUID registers and their encodings in CP15. Two of the encodings shown, with `<CRm>` == c2 and `<opc2>` == {6,7}, are reserved for future expansion of the CPUID scheme. In addition, all CP15 c0 encodings with `<CRm>` == {c3-c7} and `<opc2>` == {0-7} are reserved for future expansion of the scheme. These reserved encodings must be RAZ.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B5.1 CPUID 方式の概要

ARMv7 以前の ARM アーキテクチャのバージョンでは、アーキテクチャのバージョンがメイン ID レジスタのアーキテクチャフィールドで示されます。次のセクションを参照して下さい。

- *c0、メイン ID レジスタ (MIDR)* P. B3-81（VMSA 実装の場合）
- *c0、メイン ID レジスタ (MIDR)* P. B4-32（PMSA 実装の場合）

ARMv7 およびそれ以降では、CP15 c0 の複数のレジスタを使用する拡張プロセッサ識別方式が、アーキテクチャで実装されます。ARMv7 では、この方式の使用が義務付けられており、この方式が使用されていることは、メイン ID レジスタのアーキテクチャフィールドの値が `0xF` であることによって示されます。

注

一部の ARMv6 プロセッサでは、この方式がアーキテクチャで正式に採用される前に実装されています。

CPUID 方式では、実装されている次のような機能に関する情報が提供されます。

- プロセッサ機能
- デバッグ機能
- 補助機能、特に、実装定義の機能
- メモリモデル機能
- 命令セット機能

次のセクションでは、CPUID レジスタについてより詳しく説明します。

- *CPUID レジスタの編成*
- *CPUID レジスタの一般的な特徴* P. B5-3

レジスタの詳細な説明については、*CPUID レジスタ* P. B5-4 を参照して下さい。

本章では、アドバンスト SIMD または VFP 実装に関する識別レジスタについても説明します。これらのレジスタは、CP 10 および CP 11 にある、アドバンスト SIMD 拡張機能と VFP 拡張機能の共有レジスタ空間に存在します。これらのレジスタについては、*アドバンスト SIMD および VFP 機能識別レジスタ* P. B5-34 で説明します。

B5.1.1 CPUID レジスタの編成

CP15 の CPUID レジスタと、それらのエンコードを P. B5-3 の図 B5-1 に示します。`<CRm>` == c2 かつ `<opc2>` == {6,7} を使用して示されている 2 つのエンコードは、CPUID 方式の将来の拡張のために予約されています。さらに、`<CRm>` == {c3-c7} かつ `<opc2>` == {0-7} を使用するすべての CP15 c0 エンコードも、この方式の将来の拡張のために予約されています。これらの予約済みのエンコードは RAZ にする必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure B5-1 The CPUID register encodings

B5.1.2 General features of the CPUID registers

All of the CPUID registers are:

- 32-bit read-only registers
- accessible only in privileged modes
- when the Security Extensions are implemented, Common registers, see *Common CP15 registers* on page B3-74.

Each register is divided into eight 4-bit fields, and the possible field values are defined individually for each field. Some registers do not use all of these fields.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

図 B5- 1 CPUID レジスタのエンコード

B5.1.2 CPUID レジスタの一般的な特徴

すべての CPUID レジスタに共通する特徴は次のとおりです。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、共通レジスタです（*共通の CP15 レジスタ* P. B3- 74 を参照）。

各レジスタは、8 つの 4 ビットフィールドに分割されており、フィールドの可能な値はフィールドごとに個別に定義されます。レジスタによっては、これらのフィールドの一部しか使用されない場合があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B5.2 The CPUID registers

The CPUID registers are described in detail in the following sections:

- *CP15 c0, Processor Feature registers*
- *c0, Debug Feature Register 0 (ID_DFR0)* on page B5-6
- *c0, Auxiliary Feature Register 0 (ID_AFR0)* on page B5-8
- *CP15 c0, Memory Model Feature registers* on page B5-9
- *CP15 c0, Instruction Set Attribute registers* on page B5-19.

See also *General features of the CPUID registers* on page B5-3.

B5.2.1 CP15 c0, Processor Feature registers

The Processor Feature registers, ID_PFR0 and ID_PFR1, provide information about the instruction set state support and programmers' model for the processor. There are two Processor Feature registers, described in:

- *c0, Processor Feature Register 0 (ID_PFR0)*
- *c0, Processor Feature Register 1 (ID_PFR1)* on page B5-5
- *Accessing the Processor Feature registers* on page B5-6.

c0, Processor Feature Register 0 (ID_PFR0)

The format of ID_PFR0 is:

31 … 16	15 … 12	11 … 8	7 … 4	3 … 0
Reserved, RAZ	State3	State2	State1	State0

Bits [31:16] Reserved, RAZ.

State3, bits [15:12]

ThumbEE instruction set support. Permitted values are:

0b0000 Not supported.

0b0001 ThumbEE instruction set supported.

The value of 0b0001 is only permitted when State1 == 0b0011.

State2, bits [11:8]

Jazelle extension support. Permitted values are:

0b0000 Not supported.

0b0001 Support for Jazelle extension, without clearing of JOSCR.CV on exception entry.

0b0010 Support for Jazelle extension, with clearing of JOSCR.CV on exception entry.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B5.2 CPUID レジスタ

次のセクションでは、CPUID レジスタについて詳しく説明します。

- *CP15 c0、プロセッサ機能レジスタ*
- *c0、デバッグ機能レジスタ 0 (ID_DFR0)* P. B5-6
- *c0、補助機能レジスタ 0 (ID_AFR0)* P. B5-8
- *CP15 c0、メモリモデル機能レジスタ* P. B5-9
- *CP15 c0、命令セット属性レジスタ* P. B5-19

CPUID レジスタの一般的な特徴 P. B5-3 も参照して下さい。

B5.2.1 CP15 c0、プロセッサ機能レジスタ

プロセッサ機能レジスタ ID_PFR0 および ID_PFR1 は、プロセッサの命令セット状態のサポートとプログラマモデルについての情報を提供します。プロセッサ機能レジスタは 2 つ存在します。それぞれのレジスタについて、次のセクションで説明します。

- *c0、プロセッサ機能レジスタ 0 (ID_PFR0)*
- *c0、プロセッサ機能レジスタ 1 (ID_PFR1)* P. B5-5
- *プロセッサ機能レジスタへのアクセス* P. B5-6

c0、プロセッサ機能レジスタ 0 (ID_PFR0)

ID_PFR0 のフォーマットは次のとおりです。

31 … 16	15 … 12	11 … 8	7 … 4	3 … 0
予約、RAZ	State3	State2	State1	State0

ビット [31:16] 予約、RAZ

State3、ビット [15:12]

ThumbEE 命令セットのサポート。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 ThumbEE 命令セットがサポートされています。

0b0001 の値は、State1 == 0b0011 の場合にのみ許可されます。

State2、ビット [11:8]

Jazelle 拡張機能のサポート。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 例外開始時に JOSCR.CV をクリアしない Jazelle 拡張機能がサポートされています。

0b0010 例外開始時に JOSCR.CV をクリアする Jazelle 拡張機能がサポートされています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

State1, bits [7:4]

Thumb instruction set support. Permitted values are:

0b0000 No support for Thumb instruction set.

0b0001 Support for Thumb encoding before the introduction of Thumb-2 technology:

- all instructions are 16-bit
- a BL or BLX is a pair of 16-bit instructions
- 32-bit instructions other than BL and BLX cannot be encoded.

0b0010 Reserved.

0b0011 Support for Thumb encoding after the introduction of Thumb-2 technology, and for all 16-bit and 32-bit Thumb basic instructions.

State0, bits [3:0]

ARM instruction set support. Permitted values are:

0b0000 No support for ARM instruction set.

0b0001 Support for ARM instruction set.

c0, Processor Feature Register 1 (ID_PFR1)

The format of ID_PFR1 is:

Bits [31:12] Reserved, RAZ.

M profile programmers' model, bits [11:8]

Permitted values are:

0b0000 Not supported.

0b0010 Support for two-stack programmers' model.

The value of 0b0001 is reserved.

Security Extensions, bits [7:4]

Permitted values are:

0b0000 Not supported.

0b0001 Support for the Security Extensions.
This includes support for Monitor mode and the SMC instruction.

0b0010 As for 0b0001, and adds the ability to set the NSACR.RFR bit.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

State1、ビット [7:4]

Thumb 命令セットのサポート。許される値は次のとおりです。

0b0000 Thumb 命令セットがサポートされています。

0b0001 Thumb-2 テクノロジ導入前の Thumb エンコードがサポートされています。

- すべての命令が 16 ビットである。
- BL または BLX が 16 ビット命令のペアである。
- BL と BLX を除く 32 ビット命令はエンコードできない。

0b0010 予約

0b0011 Thumb-2 テクノロジ導入後の Thumb エンコードと、16 ビットおよび 32 ビットの Thumb 基本命令のすべてがサポートされています。

State0、ビット [3:0]

ARM 命令セットのサポート。許される値は次のとおりです。

0b0000 ARM 命令セットがサポートされていません。

0b0001 ARM 命令セットがサポートされています。

c0、プロセッサ機能レジスタ 1 (ID_PFR1)

ID_PFR1 のフォーマットは次のとおりです。

ビット [31:12] 予約、RAZ

M プロファイル プログラマモデル、ビット [11:8]

許される値は次のとおりです。

0b0000 サポートされていません。

0b0010 2 スタック プログラマモデルがサポートされています。

0b0001 の値は予約されています。

セキュリティ拡張機能、ビット [7:4]

許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 セキュリティ拡張機能がサポートされています。
これには、モニタモードと SMC 命令のサポートも含まれます。

0b0010 0b0001 と同じ機能に加えて、NSACR.RFR ビットをセットする機能が追加されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Programmers' model, bits [3:0]

Support for the standard programmers' model for ARMv4 and later. Model must support User, FIQ, IRQ, Supervisor, Abort, Undefined and System modes. Permitted values are:

0b0000 Not supported.

0b0001 Supported.

Accessing the Processor Feature registers

To access the Processor Feature Registers you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c1, and `<opc2>` set to:

- 0 for ID_PFR0
- 1 for ID_PFR1.

For example:

```
MRC p15, 0, <Rt>, c0, c1, 0    ; Read Processor Feature Register 0
```

B5.2.2 c0, Debug Feature Register 0 (ID_DFR0)

The Debug Feature Register 0, ID_DFR0, provides top level information about the debug system for the processor. You can obtain more information from the debug infrastructure, see *Debug identification registers* on page C10-3.

The format of the ID_DFR0 is:

Bits [31:24] Reserved, RAZ.

Debug model, M profile, bits [23:20]

Support for memory-mapped debug model for M profile processors. Permitted values are:

0b0000 Not supported.

0b0001 Support for M profile Debug architecture, with memory-mapped access.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

プログラマモデル、ビット [3:0]

ARMv4 およびそれ以降の標準プログラマモデルのサポート。モデルで、ユーザ、FIQ、IRQ、スーパバイザ、アボート、未定義、システムの各モードをサポートする必要があります。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされています。

プロセッサ機能レジスタへのアクセス

プロセッサ機能レジスタにアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c1、`<opc2>` を次のように設定して、CP15 レジスタを読み出します。

- ID_PFR0 の場合、0
- ID_PFR1 の場合、1

次に例を示します。

```
MRC p15, 0, <Rt>, c0, c1, 0    ; Read Processor Feature Register 0
```

B5.2.2 c0、デバッグ機能レジスタ 0 (ID_DFR0)

デバッグ機能レジスタ 0 (ID_DFR0) は、プロセッサのデバッグシステムに関するトップレベルの情報を提供します。デバッグインフラストラクチャから、さらに情報を入手することができます。*デバッグ識別レジスタ* P. C10-3 を参照して下さい。

ID_DFR0 のフォーマットは次のとおりです。

ビット [31:24] 予約、RAZ

デバッグモデル、M プロファイル、ビット [23:20]

M プロファイルプロセッサの、メモリマップされたデバッグモデルのサポート。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 メモリマップされたアクセスを含む、M プロファイル デバッグアーキテクチャがサポートされています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Memory-mapped trace model, bits [19:16]

Support for memory-mapped trace model. Permitted values are:

0b0000 Not supported.

0b0001 Support for ARM trace architecture, with memory-mapped access.
The ID register, register `0x079`, gives more information about the implementation. See also *Trace* on page C1-5.

Coprocessor trace model, bits [15:12]

Support for coprocessor-based trace model. Permitted values are:

0b0000 Not supported.

0b0001 Support for ARM trace architecture, with CP14 access.
The ID register, register `0x079`, gives more information about the implementation. See also *Trace* on page C1-5.

Memory-mapped debug model, A and R profiles, bits [11:8]

Support for memory-mapped debug model, for A and R profile processors. Permitted values are:

0b0000 Not supported, or pre-ARMv6 implementation.

0b0100 Support for v7 Debug architecture, with memory-mapped access.

Values 0b0001, 0b0010, and 0b0011 are reserved.

Coprocessor Secure debug model, bits [7:4]

Support for coprocessor-based Secure debug model, for an A profile processor that includes the Security Extensions. Permitted values are:

0b0000 Not supported.

0b0011 Support for v6.1 Debug architecture, with CP14 access.

0b0100 Support for v7 Debug architecture, with CP14 access.

Values 0b0001 and 0b0010 are reserved.

Coprocessor debug model, bits [3:0]

Support for coprocessor based debug model, for A and R profile processors. Permitted values are:

0b0000 Not supported.

0b0010 Support for v6 Debug architecture, with CP14 access.

0b0011 Support for v6.1 Debug architecture, with CP14 access.

0b0100 Support for v7 Debug architecture, with CP14 access.

Value 0b0001 is reserved.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

メモリマップされたトレースモデル、ビット [19:16]

メモリマップされたトレースモデルのサポート。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 メモリマップされたアクセスを含む、ARM トレースアーキテクチャがサポートされています。
ID レジスタのレジスタ `0x079` によって、さらに実装に関する情報が提供されます。*トレース* P. C1-5 も参照して下さい。

コプロセッサ トレースモデル、ビット [15:12]

コプロセッサベースのトレースモデルのサポート。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 CP14 アクセスを含む、ARM トレースアーキテクチャがサポートされています。
ID レジスタのレジスタ `0x079` によって、さらに実装に関する情報が提供されます。*トレース* P. C1-5 も参照して下さい。

メモリマップされたデバッグモデル、A および R プロファイル、ビット [11:8]

A および R プロファイルプロセッサの、メモリマップされたデバッグモデルのサポート。許される値は次のとおりです。

0b0000 サポートされていない、または ARMv6 以前の実装です。

0b0100 メモリマップされたアクセスを含む、v7 デバッグアーキテクチャがサポートされています。

0b0001、0b0010、0b0011 の値は予約されています。

コプロセッサ セキュアデバッグモデル、ビット [7:4]

セキュリティ拡張機能を含む A プロファイルプロセッサ用の、コプロセッサベースのセキュア デバッグモデルのサポート。許される値は次のとおりです。

0b0000 サポートされていません。

0b0011 CP14 アクセスを含む、v6.1 デバッグアーキテクチャがサポートされています。

0b0100 CP14 アクセスを含む、v7 デバッグアーキテクチャがサポートされています。

0b0001 と 0b0010 の値は予約されています。

コプロセッサ デバッグモデル、ビット [3:0]

A および R プロファイルプロセッサの、コプロセッサベースのデバッグモデルのサポート。許される値は次のとおりです。

0b0000 サポートされていません。

0b0010 CP14 アクセスを含む、v6 デバッグアーキテクチャがサポートされています。

0b0011 CP14 アクセスを含む、v6.1 デバッグアーキテクチャがサポートされています。

0b0100 CP14 アクセスを含む、v7 デバッグアーキテクチャがサポートされています。

0b0001 の値は予約されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the ID_DFR0

To access the ID_DFR0 you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c1, and `<opc2>` set to 2. For example:

```
MRC p15, 0, <Rt>, c0, c1, 2    ; Read Debug Feature Register 0
```

B5.2.3 c0, Auxiliary Feature Register 0 (ID_AFR0)

The Auxiliary Feature Register 0, ID_AFR0, provides information about the IMPLEMENTATION DEFINED features of the processor.

The format of the ID_AFR0 is:

31 … 16	15 … 12	11 … 8	7 … 4	3 … 0
Reserved, RAZ	IMP	IMP	IMP	IMP

Bits [31:16] Reserved, RAZ.

IMPLEMENTATION DEFINED, bits [15:12]

IMPLEMENTATION DEFINED, bits [11:8]

IMPLEMENTATION DEFINED, bits [7:4]

IMPLEMENTATION DEFINED, bits [3:0]

The Auxiliary Feature Register 0 has four 4-bit IMPLEMENTATION FIELDS. These fields are defined by the implementer of the design. The implementer is identified by the Implementer field of the Main ID Register, see:

- *c0, Main ID Register (MIDR)* on page B3-81, for a VMSA implementation
- *c0, Main ID Register (MIDR)* on page B4-32, for a PMSA implementation.

The Auxiliary Feature Register 0 enables implementers to include additional design features in the CPUID scheme. Field definitions for the Auxiliary Feature Register 0 might:

- differ between different implementers
- be subject to change
- migrate over time, for example if they are incorporated into the main architecture.

Accessing the ID_AFR0

To access the ID_AFR0 you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c1, and `<opc2>` set to 3. For example:

```
MRC p15, 0, <Rt>, c0, c1, 3    ; Read Auxiliary Feature Register 0
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ID_DFR0 へのアクセス

ID_DFR0 にアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c1、`<opc2>` を 2 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15, 0, <Rt>, c0, c1, 2    ; Read Debug Feature Register 0
```

B5.2.3 c0、補助機能レジスタ 0 (ID_AFR0)

補助機能レジスタ 0 (ID_AFR0) は、プロセッサの実装定義の機能についての情報を提供します。

ID_AFR0 のフォーマットは次のとおりです。

31 … 16	15 … 12	11 … 8	7 … 4	3 … 0
予約、RAZ	State3	State2	State1	State0

ビット [31:16] 予約、RAZ

実装定義、ビット [15:12]

実装定義、ビット [11:8]

実装定義、ビット [7:4]

実装定義、ビット [3:0]

補助機能レジスタ 0 には、4 つの 4 ビット実装フィールドがあります。これらのフィールドは、設計の実装者によって定義されます。実装者は、メイン ID レジスタの実装者フィールドによって識別されます。次のセクションを参照して下さい。

- *c0、メイン ID レジスタ (MIDR)* P. B3-81（VMSA 実装の場合）
- *c0、メイン ID レジスタ (MIDR)* P. B4-32（PMSA 実装の場合）

補助機能レジスタ 0 を使用すれば、実装者は CPUID 方式に新たな設計機能を含めることができます。補助機能レジスタ 0 のフィールドの定義には、次のような特徴があります。

- 実装者によって異なる。
- 変化の対象である。
- 期間が経過すると、メインのアーキテクチャに組み込まれるなどによって移行が行われる。

ID_AFR0 へのアクセス

ID_AFR0 にアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c1、`<opc2>` を 3 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15, 0, <Rt>, c0, c1, 3    ; Read Auxiliary Feature Register 0
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B5.2.4 CP15 c0, Memory Model Feature registers

The Memory Model Feature registers, ID_MMF0 to ID_MMFR3, provide general information about the implemented memory model and memory management support, including the supported cache and TLB operations. There are four Memory Model Feature registers, described in:

- *c0, Memory Model Feature Register 0 (ID_MMFR0)*
- *c0, Memory Model Feature Register 1 (ID_MMFR1)* on page B5-11
- *c0, Memory Model Feature Register 2 (ID_MMFR2)* on page B5-14
- *c0, Memory Model Feature Register 3 (ID_MMFR3)* on page B5-17
- *Accessing the Memory Model Feature registers* on page B5-19.

c0, Memory Model Feature Register 0 (ID_MMFR0)

The format of the ID_MMFR0 is:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
Innermost shareability	FCSE support	Auxiliary registers	TCM support	Shareability levels	Outermost shareability	PMSA support	VMSA support

Innermost shareability, bits [31:28]

Indicates the innermost shareability domain implemented. Permitted values are:

0b0000 Implemented as Non-cacheable.

0b0001 Implemented with hardware coherency support.

0b1111 Shareability ignored.

This field is valid only if more than one level of shareability is implemented, as indicated by the value of the Shareability levels field, bits [15:12].

When the Shareability level field is zero, this field is UNK.

FCSE support, bits [27:24]

Indicates whether the implementation includes the FCSE. Permitted values are:

0b0000 Not supported.

0b0001 Support for FCSE.

The value of 0b0001 is only permitted when the VMSA_support field has a value greater than 0b0010.

Auxiliary registers, bits [23:20]

Indicates support for Auxiliary registers. Permitted values are:

0b0000 None supported.

0b0001 Support for Auxiliary Control Register only.

0b0010 Support for Auxiliary Fault Status Registers (AIFSR and ADFSR) and Auxiliary Control Register.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B5.2.4 CP15 c0、メモリモデル機能レジスタ

メモリモデル機能レジスタ (ID_MMF0 ～ ID_MMFR3) は、サポートされているキャッシュおよび TLB 操作を含む、実装されているメモリモデルとメモリ管理のサポートについての一般的な情報を提供します。4 つのメモリモデル機能レジスタがあり、これらのレジスタについては次のセクションで説明します。

- *c0、メモリモデル機能レジスタ 0 (ID_MMFR0)*
- *c0、メモリモデル機能レジスタ 1 (ID_MMFR1)* P. B5- 11
- *c0、メモリモデル機能レジスタ 2 (ID_MMFR2)* P. B5- 14
- *c0、メモリモデル機能レジスタ 3 (ID_MMFR3)* P. B5- 17
- *メモリモデル機能レジスタへのアクセス* P. B5- 19

c0、メモリモデル機能レジスタ 0 (ID_MMFR0)

ID_MMFR0 のフォーマットは次のとおりです。

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
最も内側の共有可能性	FCSE サポート	補助レジスタ	TCM サポート	共有可能レベル	最も外側の共有可能性	PMSA サポート	VMSA サポート

最も内側の共有可能性、ビット [31:28]

実装されている、最も内側の共有可能ドメインを示します。許される値は次のとおりです。

0b0000 キャッシュ不可として実装されています。

0b0001 ハードウェアコヒーレンシ サポートとともに実装されています。

0b1111 共有可能性が無視されます。

このフィールドは、共有可能レベルフィールド（ビット [15:12]）の値によって、複数レベルの共有可能性が実装されている場合にのみ有効です。

共有可能レベルフィールドが 0 の場合、このフィールドは UNK です。

FCSE サポート、ビット [27:24]

実装に FCSE が含まれているかどうかを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 FCSE がサポートされています。

0b0001 の値は、VMSA_support フィールドに 0b0010 を超える値が設定されている場合にのみ許可されます。

補助レジスタ、ビット [23:20]

補助レジスタのサポートを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 補助制御レジスタのみがサポートされています。

0b0010 補助フォールトステータス レジスタ（AIFSR および ADFSR）と補助制御レジスタがサポートされています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

TCM support, bits [19:16]

Indicates support for TCMs and associated DMAs. Permitted values are:

0b0000 Not supported.

0b0001 Support is IMPLEMENTATION DEFINED. ARMv7 requires this setting.

0b0010 Support for TCM only, ARMv6 implementation.

0b0011 Support for TCM and DMA, ARMv6 implementation.

Note

An ARMv7 implementation might include an ARMv6 model for TCM support. However, in ARMv7 this is an IMPLEMENTATION DEFINED option, and therefore it must be represented by the 0b0001 encoding in this field.

Shareability levels, bits [15:12]

Indicates the number of shareability levels implemented. Permitted values are:

0b0000 One level of shareability implemented.

0b0001 Two levels of shareability implemented.

Outermost shareability, bits [11:8]

Indicates the outermost shareability domain implemented. Permitted values are:

0b0000 Implemented as Non-cacheable.

0b0001 Implemented with hardware coherency support.

0b1111 Shareability ignored.

PMSA support, bits [7:4]

Indicates support for a PMSA. Permitted values are:

0b0000 Not supported.

0b0001 Support for IMPLEMENTATION DEFINED PMSA.

0b0010 Support for PMSAv6, with a Cache Type Register implemented.

0b0011 Support for PMSAv7, with support for memory subsections. ARMv7-R profile.

When the PMSA support field is set to a value other than 0b0000 the VMSA support field must be set to 0b0000.

VMSA support, bits [3:0]

Indicates support for a VMSA. Permitted values are:

0b0000 Not supported.

0b0001 Support for IMPLEMENTATION DEFINED VMSA.

0b0010 Support for VMSAv6, with Cache and TLB Type Registers implemented.

0b0011 Support for VMSAv7, with support for remapping and the access flag. ARMv7-A profile.

When the VMSA support field is set to a value other than 0b0000 the PMSA support field must be set to 0b0000.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

TCM サポート、ビット [19:16]

TCM と、関連する DMA のサポートを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートは実装定義です。ARMv7 ではこの設定が必須です。

0b0010 TCM のみがサポートされています（ARMv6 実装）。

0b0011 TCM と DMA がサポートされています（ARMv6 実装）。

注

ARMv7 実装に、TCM サポート用の ARMv6 モデルが含まれている場合があります。ただし、ARMv7 では、これが実装定義オプションであるため、このフィールドの 0b0001 エンコードによって示す必要があります。

共有可能レベル、ビット [15:12]

実装されている共有可能レベル数を示します。許される値は次のとおりです。

0b0000 1 レベルの共有可能性が実装されています。

0b0001 2 レベルの共有可能性が実装されています。

最も外側の共有可能性、ビット [11:8]

実装されている、最も外側の共有可能ドメインを示します。許される値は次のとおりです。

0b0000 キャッシュ不可として実装されています。

0b0001 ハードウェアコヒーレンシ サポートとともに実装されています。

0b1111 共有可能性が無視されます。

PMSA サポート、ビット [7:4]

PMSA のサポートを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 実装定義の PMSA がサポートされています。

0b0010 PMSAv6 がサポートされ、キャッシュタイプ レジスタが実装されています。

0b0011 PMSAv7 がサポートされ、メモリサブセクションがサポートされています。ARMv7-R プロファイル。

PMSA サポートフィールドを 0b0000 以外の値に設定した場合は、VMSA サポートフィールドを 0b0000 に設定する必要があります。

VMSA サポート、ビット [3:0]

VMSA のサポートを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 実装定義の VMSA がサポートされています。

0b0010 VMSAv6 がサポートされ、キャッシュタイプ レジスタと TLB タイプレジスタが実装されています。

0b0011 VMSAv7 がサポートされ、再マッピングとアクセスフラグがサポートされています。ARMv7-A プロファイル。

VMSA サポートフィールドを 0b0000 以外の値に設定した場合は、PMSA サポートフィールドを 0b0000 に設定する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c0, Memory Model Feature Register 1 (ID_MMFR1)

The format of the ID_MMFR1 is:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
Branch Predictor	L1 cache Test and Clean	L1 unified cache	L1 Harvard cache	L1 unified cache s/w	L1 Harvard cache s/w	L1 unified cache VA	L1 Harvard cache VA

Branch predictor, bits [31:28]

Indicates branch predictor management requirements. Permitted values are:

0b0000 No branch predictor, or no MMU present. Implies a fixed MPU configuration.

0b0001 Branch predictor requires flushing on:

- enabling or disabling the MMU
- writing new data to instruction locations
- writing new mappings to the translation tables
- any change to the TTBR0, TTBR1, or TTBCR registers
- changes of FCSE ProcessID or ContextID.

0b0010 Branch predictor requires flushing on:

- enabling or disabling the MMU
- writing new data to instruction locations
- writing new mappings to the translation tables
- any change to the TTBR0, TTBR1, or TTBCR registers without a corresponding change to the FCSE ProcessID or ContextID.

0b0011 Branch predictor requires flushing only on:

- writing new data to instruction locations.

0b0100 For execution correctness, branch predictor requires no flushing at any time.

Note

The branch predictor is described in some documentation as the Branch Target Buffer.

L1 cache Test and Clean, bits [27:24]

Indicates the supported Level 1 data cache test and clean operations, for Harvard or unified cache implementations. Permitted values are:

0b0000 None supported. This is the required setting for ARMv7.

0b0001 Supported Level 1 data cache test and clean operations are:

- Test and clean data cache.

0b0010 As for 0b0001, and adds:

- Test, clean, and invalidate data cache.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c0、メモリモデル機能レジスタ 1（ID_MMFR1）

ID_MMFR1 のフォーマットは次のとおりです。

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
分岐予測器	L1キャッシュのテストとクリーニング	L1統一キャッシュ	L1ハーバードキャッシュ	L1統一キャッシュs/w	L1ハーバードキャッシュs/w	L1統一キャッシュVA	L1ハーバードキャッシュVA

分岐予測器、ビット [31:28]

分岐予測器の管理要件を示します。許される値は次のとおりです。

0b0000 分岐予測器または MMU が存在しません。固定 MPU 構成を意味します。

0b0001 次の状況で、分岐予測器をフラッシュする必要があります。

- MMU の稼動または非稼動
- 命令位置への新しいデータの書き込み
- 変換テーブルへの新しいマッピングの書き込み
- TTBR0、TTBR1、または TTBCR レジスタの変更
- FCSE ProcessID または ContextID の変更

0b0010 次の状況で、分岐予測器をフラッシュする必要があります。

- MMU の稼動または非稼動
- 命令位置への新しいデータの書き込み
- 変換テーブルへの新しいマッピングの書き込み
- FCSE ProcessID または ContextID の対応する変更を伴わない、TTBR0、TTBR1、または TTBCR レジスタの変更

0b0011 次の状況でのみ、分岐予測器をフラッシュする必要があります。

- 命令位置への新しいデータの書き込み

0b0100 実行の正確性を保証するために、分岐予測器のフラッシュを行う必要はありません。

注

一部の説明書では、分岐予測器が分岐先バッファと記載されています。

L1 キャッシュのテストとクリーニング、ビット [27:24]

ハーバードまたは統一キャッシュ実装について、サポートされているレベル 1 データキャッシュのテスト操作とクリーニング操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。ARMv7 では、これが必須設定です。

0b0001 サポートされているレベル 1 データキャッシュのテスト操作とクリーニング操作は次のとおりです。

- データキャッシュのテストとクリーニング

0b0010 0b0001 の操作に加えて、次の操作が追加されます。

- データキャッシュのテスト、クリーニング、および無効化

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

L1 unified cache, bits [23:20]

Indicates the supported entire Level 1 cache maintenance operations, for a unified cache implementation. Permitted values are:

0b0000 None supported. This is the required setting for ARMv7, because ARMv7 requires a hierarchical cache implementation.

0b0001 Supported entire Level 1 cache operations are:

- Invalidate cache, including branch predictor if appropriate
- Invalidate branch predictor, if appropriate.

0b0010 As for 0b0001, and adds:

- Clean cache. Uses a recursive model, using the cache dirty status bit.
- Clean and invalidate cache. Uses a recursive model, using the cache dirty status bit.

If this field is set to a value other than 0b0000 then the L1 Harvard cache field, bits [19:16], must be set to 0b0000.

L1 Harvard cache, bits [19:16]

Indicates the supported entire Level 1 cache maintenance operations, for a Harvard cache implementation. Permitted values are:

0b0000 None supported. This is the required setting for ARMv7, because ARMv7 requires a hierarchical cache implementation.

0b0001 Supported entire Level 1 cache operations are:

- Invalidate instruction cache, including branch predictor if appropriate
- Invalidate branch predictor, if appropriate.

0b0010 As for 0b0001, and adds:

- Invalidate data cache
- Invalidate data cache and instruction cache, including branch predictor if appropriate.

0b0011 As for 0b0010, and adds:

- Clean data cache. Uses a recursive model, using the cache dirty status bit.
- Clean and invalidate data cache. Uses a recursive model, using the cache dirty status bit.

If this field is set to a value other than 0b0000 then the L1 unified cache field, bits [23:20], must be set to 0b0000.

L1 unified cache s/w, bits [15:12]

Indicates the supported Level 1 cache line maintenance operations by set/way, for a unified cache implementation. Permitted values are:

0b0000 None supported. This is the required setting for ARMv7, because ARMv7 requires a hierarchical cache implementation.

0b0001 Supported Level 1 unified cache line maintenance operations by set/way are:

- Clean cache line by set/way.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

L1 統一キャッシュ、ビット [23:20]

統一キャッシュ実装について、サポートされているレベル 1 キャッシュ全体の保守操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。ARMv7 では、階層型キャッシュ実装が必要なため、これが必須設定です。

0b0001 サポートされているレベル 1 キャッシュ全体の操作は次のとおりです。

- 分岐予測器の無効化（該当する場合）を含む、キャッシュの無効化
- 分岐予測器の無効化（該当する場合）

0b0010 0b0001 の操作に加えて、次の操作が追加されます。

- キャッシュのクリーニング。キャッシュダーティステータス ビットによる再帰的モデルが使用されます。
- キャッシュのクリーンニングと無効化。キャッシュダーティステータス ビットによる再帰的モデルが使用されます。

このフィールドを 0b0000 以外の値に設定した場合は、L1 ハーバードキャッシュフィールド（ビット [19:16]）を 0b0000 に設定する必要があります。

L1 ハーバードキャッシュ、ビット [19:16]

ハーバードキャッシュ実装について、サポートされているレベル 1 キャッシュ全体の保守操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。ARMv7 では、階層型キャッシュ実装が必要なため、これが必須設定です。

0b0001 サポートされているレベル 1 キャッシュ全体の操作は次のとおりです。

- 分岐予測器の無効化（該当する場合）を含む、命令キャッシュの無効化
- 分岐予測器の無効化（該当する場合）

0b0010 0b0001 の操作に加えて、次の操作が追加されます。

- データキャッシュの無効化
- 分岐予測器の無効化（該当する場合）を含む、データキャッシュと命令キャッシュの無効化

0b0011 0b0010 の操作に加えて、次の操作が追加されます。

- データキャッシュのクリーニング。キャッシュダーティステータス ビットによる再帰的モデルが使用されます。
- データキャッシュのクリーニングと無効化。キャッシュダーティステータス ビットによる再帰的モデルが使用されます。

このフィールドを 0b0000 以外の値に設定した場合は、L1 統一キャッシュフィールド（ビット [23:20]）を 0b0000 に設定する必要があります。

L1 統一キャッシュ s/w、ビット [15:12]

統一キャッシュ実装について、サポートされている、セット / ウェイによるレベル 1 キャッシュラインの保守操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。ARMv7 では、階層型キャッシュ実装が必要なため、これが必須設定です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b0010 As for 0b0001, and adds:

- Clean and invalidate cache line by set/way.

0b0011 As for 0b0010, and adds:

- Invalidate cache line by set/way.

If this field is set to a value other than 0b0000 then the L1 Harvard cache s/w field, bits [11:8], must be set to 0b0000.

L1 Harvard cache s/w, bits [11:8]

Indicates the supported Level 1 cache line maintenance operations by set/way, for a Harvard cache implementation. Permitted values are:

0b0000 None supported. This is the required setting for ARMv7, because ARMv7 requires a hierarchical cache implementation.

0b0001 Supported Level 1 Harvard cache line maintenance operations by set/way are:

- Clean data cache line by set/way
- Clean and invalidate data cache line by set/way.

0b0010 As for 0b0001, and adds:

- Invalidate data cache line by set/way.

0b0011 As for 0b0010, and adds:

- Invalidate instruction cache line by set/way.

If this field is set to a value other than 0b0000 then the L1 unified cache s/w field, bits [15:12], must be set to 0b0000.

L1 unified cache VA, bits [7:4]

Indicates the supported Level 1 cache line maintenance operations by MVA, for a unified cache implementation. Permitted values are:

0b0000 None supported. This is the required setting for ARMv7, because ARMv7 requires a hierarchical cache implementation.

0b0001 Supported Level 1 unified cache line maintenance operations by MVA are:

- Clean cache line by MVA
- Invalidate cache line by MVA
- Clean and invalidate cache line by MVA.

0b0010 As for 0b0001, and adds:

- Invalidate branch predictor by MVA, if branch predictor is implemented.

If this field is set to a value other than 0b0000 then the L1 Harvard cache VA field, bits [3:0], must be set to 0b0000.

L1 Harvard cache VA, bits [3:0]

Indicates the supported Level 1 cache line maintenance operations by MVA, for a Harvard cache implementation. Permitted values are:

0b0000 None supported. This is the required setting for ARMv7, because ARMv7 requires a hierarchical cache implementation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b0001 サポートされている、セット / ウェイによるレベル 1 統一キャッシュラインの保守操作は次のとおりです。

- セット / ウェイによるキャッシュラインのクリーニング

0b0010 0b0001 の操作に加えて、次の操作が追加されます。

- セット / ウェイによるキャッシュラインのクリーニングと無効化

0b0011 0b0010 の操作に加えて、次の操作が追加されます。

- セット / ウェイによるキャッシュラインの無効化

このフィールドを 0b0000 以外の値に設定した場合は、L1 ハーバードキャッシュ s/w フィールド（ビット [11:8]）を 0b0000 に設定する必要があります。

L1 ハーバードキャッシュ s/w、ビット [11:8]

ハーバードキャッシュ実装について、サポートされている、セット / ウェイによるレベル 1 キャッシュラインの保守操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。ARMv7 では、階層型キャッシュ実装が必要なため、これが必須設定です。

0b0001 サポートされている、セット / ウェイによるレベル 1 ハーバードキャッシュラインの保守操作は次のとおりです。

- セット / ウェイによるデータ キャッシュラインのクリーニング
- セット / ウェイによるデータ キャッシュラインのクリーニングと無効化

0b0010 0b0001 の操作に加えて、次の操作が追加されます。

- セット / ウェイによるデータ キャッシュラインの無効化

0b0011 0b0010 の操作に加えて、次の操作が追加されます。

- セット / ウェイによる命令キャッシュラインの無効化

このフィールドを 0b0000 以外の値に設定した場合は、L1 統一キャッシュ s/w フィールド（ビット [15:12]）を 0b0000 に設定する必要があります。

L1 統一キャッシュ VA、ビット [7:4]

統一キャッシュ実装について、サポートされている、MVA によるレベル 1 キャッシュラインの保守操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。ARMv7 では、階層型キャッシュ実装が必要なため、これが必須設定です。

0b0001 サポートされている、MVA によるレベル 1 統一キャッシュラインの保守操作は次のとおりです。

- MVA によるキャッシュラインのクリーニング
- MVA によるキャッシュラインの無効化
- MVA によるキャッシュラインのクリーニングと無効化

0b0010 0b0001 の操作に加えて、次の操作が追加されます。

- 分岐予測器が実装されている場合、MVA による分岐予測器の無効化

このフィールドを 0b0000 以外の値に設定した場合は、L1 ハーバードキャッシュ VA フィールド（ビット [3:0]）を 0b0000 に設定する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b0001 Supported Level 1 Harvard cache line maintenance operations by MVA are:

- Clean data cache line by MVA
- Invalidate data cache line by MVA
- Clean and invalidate data cache line by MVA
- Clean instruction cache line by MVA.

0b0010 As for 0b0001, and adds:

- Invalidate branch predictor by MVA, if branch predictor is implemented.

If this field is set to a value other than 0b0000 then the L1 unified cache VA field, bits [7:4], must be set to 0b0000.

c0, Memory Model Feature Register 2 (ID_MMFR2)

The format of the ID_MMFR2 is:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
HW access flag	WFI stall	Mem barrier	Unified TLB	Harvard TLB	L1 Harvard range	L1 Harvard bg prefetch	L1 Harvard fg prefetch

HW access flag, bits [31:28]

Indicates support for a Hardware access flag, as part of the VMSAv7 implementation. Permitted values are:

0b0000 Not supported.

0b0001 Support for VMSAv7 access flag, updated in hardware.

On an ARMv7-R implementation this field must be 0b0000.

WFI stall, bits [27:24]

Indicates the support for Wait For Interrupt (WFI) stalling. Permitted values are:

0b0000 Not supported.

0b0001 Support for WFI stalling.

Mem barrier, bits [23:20]

Indicates the supported CP15 memory barrier operations:

0b0000 None supported.

0b0001 Supported CP15 Memory barrier operations are:

- Data Synchronization Barrier (DSB). In previous versions of the ARM architecture, DSB was named Data Write Barrier (DWB).

0b0010 As for 0b0001, and adds:

- Instruction Synchronization Barrier (ISB). In previous versions of the ARM architecture, the ISB operation was called Prefetch Flush.
- Data Memory Barrier (DMB).

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

L1 ハーバードキャッシュ VA、ビット [3:0]

ハーバードキャッシュ実装について、サポートされている、MVA によるレベル 1 キャッシュラインの保守操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。ARMv7 では、階層型キャッシュ実装が必要なため、これが必須設定です。

0b0001 サポートされている、MVA によるレベル 1 ハーバード キャッシュラインの保守操作は次のとおりです。

- MVA によるデータ キャッシュラインのクリーニング
- MVA によるデータ キャッシュラインの無効化
- MVA によるデータ キャッシュラインのクリーニングと無効化
- MVA による命令キャッシュラインのクリーニング

0b0010 0b0001 の操作に加えて、次の操作が追加されます。

- 分岐予測器が実装されている場合、MVA による分岐予測器の無効化

このフィールドを 0b0000 以外の値に設定した場合は、L1 統一キャッシュ VA フィールド（ビット [7:4]）を 0b0000 に設定する必要があります。

c0、メモリモデル機能レジスタ 2 (ID_MMFR2)

ID_MMFR2 のフォーマットは次のとおりです。

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
HWアクセスフラグ	WFIストール	メモリバリア	統一TLB	ハーバードTLB	L1ハーバード範囲	L1ハーバードbgプリフェッチ	L1ハーバードfgプリフェッチ

HW アクセスフラグ、ビット [31:28]

VMSAv7 実装の一部として、ハードウェア アクセスフラグのサポートを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 ハードウェアで更新される、VMSAv7 アクセスフラグがサポートされています。

ARMv7-R 実装では、このフィールドを 0b0000 にする必要があります。

WFI ストール、ビット [27:24]

割り込み待ち (WFI) ストールのサポートを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 WFI ストールがサポートされています。

メモリバリア、ビット [23:20]

サポートされている CP15 メモリバリア操作を示します。

0b0000 サポートされていません。

0b0001 サポートされている CP15 メモリバリア操作は次のとおりです。

- データ同期バリア (DSB)。以前のバージョンの ARM アーキテクチャでは、DSB は DWB（データライト バリア）と呼ばれていました。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Unified TLB, bits [19:16]

Indicates the supported TLB maintenance operations, for a unified TLB implementation. Permitted values are:

0b0000 Not supported.

0b0001 Supported unified TLB maintenance operations are:
- Invalidate all entries in the TLB
- Invalidate TLB entry by MVA.

0b0010 As for 0b0001, and adds:
- Invalidate TLB entries by ASID match.

0b0011 As for 0b0010 and adds:
- Invalidate TLB entries by MVA All ASID.

If this field is set to a value other than 0b0000 then the Harvard TLB field, bits [15:12], must be set to 0b0000.

Harvard TLB, bits [15:12]

Indicates the supported TLB maintenance operations, for a Harvard TLB implementation. Permitted values are:

0b0000 Not supported.

0b0001 Supported Harvard TLB maintenance operations are:
- Invalidate all entries in the ITLB and the DTLB.
  This is a shared unified TLB operation.
- Invalidate all ITLB entries.
- Invalidate all DTLB entries.
- Invalidate ITLB entry by MVA.
- Invalidate DTLB entry by MVA.

0b0010 As for 0b0001, and adds:
- Invalidate ITLB and DTLB entries by ASID match.
  This is a shared unified TLB operation.
- Invalidate ITLB entries by ASID match
- Invalidate DTLB entries by ASID match.

If this field is set to a value other than 0b0000 then the Unified TLB field, bits [19:16], must be set to 0b0000.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b0010 0b0001 の操作に加えて、次の操作が追加されます。
- 命令同期バリア (ISB)。以前のバージョンの ARM アーキテクチャでは、ISB 操作はプリフェッチフラッシュと呼ばれていました。
- データ メモリバリア (DMB)

統一 TLB、ビット [19:16]

統一 TLB 実装について、サポートされる TLB 保守操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされている統一 TLB 保守操作は次のとおりです。
- TLB のすべてのエントリの無効化
- MVA による TLB エントリの無効化

0b0010 0b0001 の操作に加えて、次の操作が追加されます。
- ASID 一致による TLB エントリの無効化

0b0011 0b0010 の操作に加えて、次の操作が追加されています。
- MVA とすべての ASID による TLB エントリの無効化

このフィールドを 0b0000 以外の値に設定した場合は、ハーバード TLB フィールド（ビット [15:12]）を 0b0000 に設定する必要があります。

ハーバード TLB、ビット [15:12]

ハーバード TLB 実装について、サポートされる TLB 保守操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされているハーバード TLB 保守操作は次のとおりです。
- ITLB および DTLB のすべてのエントリの無効化
  これは、共有される統一 TLB 操作です。
- すべての ITLB エントリの無効化
- すべての DTLB エントリの無効化
- MVA による ITLB エントリの無効化
- MVA による DTLB エントリの無効化

0b0010 0b0001 の操作に加えて、次の操作が追加されます。
- ASID 一致による ITLB および DTLB ントリの無効化
  これは、共有される統一 TLB 操作です。
- ASID 一致による ITLB エントリの無効化
- ASID 一致による DTLB エントリの無効化

このフィールドを 0b0000 以外の値に設定した場合は、統一 TLB フィールド（ビット [19:16]）を 0b0000 に設定する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

L1 Harvard range, bits [11:8]

Indicates the supported Level 1 cache maintenance range operations, for a Harvard cache implementation. Permitted values are:

0b0000 Not supported.

0b0001 Supported Level 1 Harvard cache maintenance range operations are:

- Invalidate data cache range by VA
- Invalidate instruction cache range by VA
- Clean data cache range by VA
- Clean and invalidate data cache range by VA.

L1 Harvard bg prefetch, bits [7:4]

Indicates the supported Level 1 cache background prefetch operations, for a Harvard cache implementation. When supported, background prefetch operations are non-blocking operations. Permitted values are:

0b0000 Not supported.

0b0001 Supported Level 1 Harvard cache foreground prefetch operations are:

- Prefetch instruction cache range by VA
- Prefetch data cache range by VA.

L1 Harvard fg prefetch, bits [3:0]

Indicates the supported Level 1 cache foreground prefetch operations, for a Harvard cache implementation. When supported, foreground prefetch operations are blocking operations. Permitted values are:

0b0000 Not supported.

0b0001 Supported Level 1 Harvard cache foreground prefetch operations are:

- Prefetch instruction cache range by VA
- Prefetch data cache range by VA.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

L1 ハーバード範囲、ビット [11:8]

ハーバードキャッシュ実装について、サポートされているレベル 1 キャッシュ範囲の保守操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされている、レベル 1 ハーバード キャッシュ範囲の保守操作は次のとおりです。

- VA によるデータキャッシュ範囲の無効化
- VA による命令キャッシュ範囲の無効化
- VA によるデータキャッシュ範囲のクリーニング
- VA によるデータキャッシュ範囲のクリーニングと無効化

L1 ハーバード bg プリフェッチ、ビット [7:4]

ハーバードキャッシュ実装について、サポートされているレベル 1 キャッシュのバックグラウンドプリフェッチ操作を示します。サポートされているバックグラウンドプリフェッチ操作は、非ブロッキング操作です。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされている、レベル 1 ハーバード キャッシュのフォアグラウンドプリフェッチ操作は次のとおりです。

- VA による命令キャッシュ範囲のプリフェッチ
- VA によるデータキャッシュ範囲のプリフェッチ

L1 ハーバード fg プリフェッチ、ビット [3:0]

ハーバードキャッシュ実装について、サポートされているレベル 1 キャッシュのフォアグラウンドプリフェッチ操作を示します。サポートされているフォアグラウンドプリフェッチ操作は、ブロッキング操作です。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされている、レベル 1 ハーバード キャッシュのフォアグラウンドプリフェッチ操作は次のとおりです。

- VA による命令キャッシュ範囲のプリフェッチ
- VA によるデータキャッシュ範囲のプリフェッチ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c0, Memory Model Feature Register 3 (ID_MMFR3)

The format of the ID_MMFR3 is:

Supersection support, bits [31:28]

On a VMSA implementation, indicates whether Supersections are supported. Permitted values are:

0b0000 Supersections supported.

0b1111 Supersections not supported.

All other values are reserved.

Note

The sense of this identification is reversed from the normal usage in the CPUID mechanism, with the value of zero indicating that the feature is supported.

Bits [27:24] Reserved, RAZ.

Coherent walk, bits [23:20]

Indicates whether Translation table updates require a clean to the point of unification. Permitted values are:

0b0000 Updates to the translation tables require a clean to the point of unification to ensure visibility by subsequent translation table walks.

0b0001 Updates to the translation tables do not require a clean to the point of unification to ensure visibility by subsequent translation table walks.

Bits [19:16] Reserved, RAZ.

Maintenance broadcast, bits [15:12]

Indicates whether Cache, TLB and branch predictor operations are broadcast. Permitted values are:

0b0000 Cache, TLB and branch predictor operations only affect local structures.

0b0001 Cache and branch predictor operations affect structures according to shareability and defined behavior of instructions. TLB operations only affect local structures.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c0、メモリモデル機能レジスタ 3 (ID_MMFR3)

ID_MMFR3 のフォーマットは次のとおりです。

スーパーセクションサポート、ビット [31:28]

VMSA 実装で、スーパーセクションがサポートされているかどうかを示します。許される値は次のとおりです。

0b0000 スーパーセクションがサポートされています。

0b1111 スーパーセクションがサポートされていません。

他の値はすべて予約されています。

注

この識別の意味は、0 の値が機能のサポートを意味する CPUID メカニズムでの通常の使用とは逆になっています。

ビット [27:24] 予約、RAZ

コヒーレントウォーク、ビット [23:20]

変換テーブル更新により、統一時点までのクリーニングが必要かどうかを示します。許される値は次のとおりです。

0b0000 変換テーブルの更新を行った場合、統一時点までクリーニングを実行して、後続の変換テーブルウォークから可視であることを保証する必要があります。

0b0001 変換テーブルの更新を行った場合、統一時点までクリーニングを実行して、後続の変換テーブルウォークから可視であることを保証する必要はありません。

ビット [19:16] 予約、RAZ

保守ブロードキャスト、ビット [15:12]

キャッシュ、TLB、分岐予測器の操作がブロードキャストされるかどうかを示します。許される値は次のとおりです。

0b0000 キャッシュ、TLB、分岐予測器の操作は、ローカル構造にのみ影響します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b0010 Cache, TLB and branch predictor operations affect structures according to shareability an defined behavior of instructions.

BP maintain, bits [11:8]

Indicates the supported branch predictor maintenance operations in an implementation with hierarchical cache maintenance operations. Permitted values are:

0b0000 None supported.

0b0001 Supported branch predictor maintenance operations are:

- Invalidate entire branch predictor array

0b0010 As for 0b0001, and adds:

- Invalidate branch predictor by MVA.

Cache maintain s/w, bits [7:4]

Indicates the supported cache maintenance operations by set/way, in an implementation with hierarchical caches. Permitted values are:

0b0000 None supported.

0b0001 Supported hierarchical cache maintenance operations by set/way are:

- Invalidate data cache by set/way
- Clean data cache by set/way
- Clean and invalidate data cache by set/way.

In a unified cache implementation, the data cache operations apply to the unified caches.

Cache maintain MVA, bits [3:0]

Indicates the supported cache maintenance operations by MVA, in an implementation with hierarchical caches. Permitted values are:

0b0000 None supported.

0b0001 Supported hierarchical cache maintenance operations by MVA are:

- Invalidate data cache by MVA
- Clean data cache by MVA
- Clean and invalidate data cache by MVA
- Invalidate instruction cache by MVA
- Invalidate all instruction cache entries.

In a unified cache implementation, the data cache operations apply to the unified caches, and the instruction cache operations are not implemented.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

0b0001 キャッシュと分岐予測器の操作が、共有可能性と、定義された命令の動作に従って構造に影響します。TLB 操作はローカル構造にのみ影響します。

0b0010 キャッシュ、TLB、分岐予測器の操作が、共有可能性と、定義された命令動作に従って構造に影響します。

BP 保守、ビット [11:8]

階層型キャッシュ保守操作を含む実装で、サポートされている分岐予測器の保守操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされている分岐予測器の保守操作は次のとおりです。

- 分岐予測器アレイ全体の無効化

0b0010 0b0001 の操作に加えて、次の操作が追加されます。

- MVA による分岐予測器の無効化

キャッシュ保守 s/w、ビット [7:4]

階層型キャッシュを含む実装で、サポートされている、セット / ウェイによるキャッシュ保守操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされている、セット / ウェイによる階層型キャッシュ保守操作は次のとおりです。

- セット / ウェイによるデータ キャッシュの無効化
- セット / ウェイによるデータキャッシュのクリーニング
- セット / ウェイによるデータキャッシュのクリーニングと無効化

統一キャッシュ実装では、データキャッシュ操作が統一キャッシュに適用されます。

キャッシュ保守 MVA、ビット [3:0]

階層型キャッシュを含む実装で、サポートされている、MVA によるキャッシュ保守操作を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされている、MVA による階層型キャッシュ保守操作は次のとおりです。

- MVA によるデータキャッシュの無効化
- MVA によるデータキャッシュのクリーニング
- MVA によるデータキャッシュのクリーニングと無効化
- MVA による命令キャッシュの無効化
- すべての命令キャッシュエントリの無効化

統一キャッシュ実装では、データキャッシュ操作が統一キャッシュに適用され、命令キャッシュ操作は実装されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the Memory Model Feature registers

To access the Memory Model Feature Registers you read the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c0, `<CRm>` set to c1, and `<opc2>` set to:

- 4 for the ID_MMFR0
- 5 for the ID_MMFR1
- 6 for the ID_MMFR2
- 7 for the ID_MMFR3.

For example:

```
MRC p15, 0, <Rt>, c0, c1, 6    ; Read Memory Model Feature Register 2
```

B5.2.5 CP15 c0, Instruction Set Attribute registers

The Instruction Set Attribute registers, ID_ISAR0 to ID_ISAR5, provide information about the instruction set supported by the processor. The instruction set is divided into:

- The basic instructions, for the ARM, Thumb, and ThumbEE instruction sets. If the Processor Feature Register 0 indicates support for one of these instruction sets then all basic instructions that have encodings in the corresponding instruction set must be implemented.
- The non-basic instructions. The Instruction Set Attribute registers indicate which of these instructions are implemented.

Instruction set descriptions in the CPUID scheme on page B5-20 describes the division of the instruction set into basic and non-basic instructions.

Summary of Instruction Set Attribute register attributes on page B5-22 lists all of the attributes and shows which register holds each attribute.

ARMv7 implements six Instruction Set Attribute registers, described in:

- *c0, Instruction Set Attribute Register 0 (ID_ISAR0)* on page B5-24
- *c0, Instruction Set Attribute Register 1 (ID_ISAR1)* on page B5-25
- *c0, Instruction Set Attribute Register 2 (ID_ISAR2)* on page B5-27
- *c0, Instruction Set Attribute Register 3 (ID_ISAR3)* on page B5-29
- *c0, Instruction Set Attribute Register 4 (ID_ISAR4)* on page B5-31
- *c0, Instruction Set Attribute Register 5 (ID_ISAR5)* on page B5-33
- *Accessing the Instruction Set Attribute registers* on page B5-33.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

メモリモデル機能レジスタへのアクセス

メモリモデル機能レジスタにアクセスするには、`<opc1>` を 0、`<CRn>` を c0、`<CRm>` を c1、`<opc2>` を次の値に設定して、CP15 レジスタを読み出します。

- ID_MMFR0 の場合、4
- ID_MMFR1 の場合、5
- ID_MMFR2 の場合、6
- ID_MMFR3 の場合、7

次に例を示します。

```
MRC p15, 0, <Rt>, c0, c1, 6    ; Read Memory Model Feature Register 2
```

B5.2.5 CP15 c0、命令セット属性レジスタ

命令セット属性レジスタ (ID_ISAR0 ～ ID_ISAR5) は、プロセッサでサポートされている命令セットに関する情報を提供します。命令セットは次のように分類されます。

- ARM、Thumb、ThumbEE 命令セット用の基本命令。プロセッサ機能レジスタ 0 で、これらの命令セットのいずれかのサポートが示されている場合は、対応する命令セット内にエンコードを持つすべての基本命令を実装する必要があります。
- 非基本命令。命令セット属性レジスタで、これらの命令のうち、どの命令が実装されているかが示されます。

命令セットの基本命令と非基本命令への分割方法については、*CPUID 方式での命令セットの説明* P. B5-20 で説明します。

すべての属性のリストと、各属性を保持するレジスタを、*命令セット属性レジスタの属性の要約* P. B5-22 に示します。

ARMv7 には 6 つの命令セット属性レジスタがあります。次のセクションでは、これらのレジスタについて説明します。

- *c0、命令セット属性レジスタ 0 (ID_ISAR0)* P. B5-24
- *c0、命令セット属性レジスタ 1 (ID_ISAR1)* P. B5-25
- *c0、命令セット属性レジスタ 2 (ID_ISAR2)* P. B5-27
- *c0、命令セット属性レジスタ 3 (ID_ISAR3)* P. B5-29
- *c0、命令セット属性レジスタ 4 (ID_ISAR4)* P. B5-31
- *c0、命令セット属性レジスタ 5 (ID_ISAR5)* P. B5-33
- *命令セット属性レジスタへのアクセス* P. B5-33

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Instruction set descriptions in the CPUID scheme

The following subsections describe how the CPUID scheme describes the instruction set, and how instructions are classified as either basic or non-basic:

- *General rules for instruction classification*
- *Data-processing instructions*
- *Multiply instructions* on page B5-21
- *Branches* on page B5-21
- *Load or Store single word instructions* on page B5-21
- *Load or Store multiple word instructions* on page B5-21
- *Q flag support in the PSRs* on page B5-21.

General rules for instruction classification

Two general rules apply to the description of instruction classification given in this section:

1. The rules about an instruction being basic do not guarantee that it is available in any particular instruction set. For example, the rules given in this section classify `MOV R0, #123456789` as a basic instruction, but this instruction is not available in any existing ARM instruction set.

2. Whether an instruction is conditional or unconditional never makes any difference to whether it is a basic instruction.

Data-processing instructions

The data-processing instructions are:

```
ADC   ADD   AND   ASR   BIC   CMN   CMP   EOR   LSL   LSR   MOV   MVN
NEG   ORN   ORR   ROR   RRX   RSB   RSC   SBC   SUB   TEQ   TST
```

An instruction from this group is a basic instruction if these conditions both apply:

- The second source operand, or the only source operand of a `MOV` or `MVN` instruction, is an immediate or an unshifted register.

  Note

  A `MOV` instruction with a shifted register source operand must be treated as the equivalent `ASR`, `LSL`, `LSR`, `ROR`, or `RRX` instruction, see *MOV (shifted register)* on page A8-198.

- The instruction is not one of the exception return instructions described in *SUBS PC, LR and related instructions* on page B6-25.

If either of these conditions does not apply then the instruction is a non-basic instruction. One or both of these attributes in the Instruction Set Attribute registers shows the support for non-basic data-processing instructions:

- PSR_instrs
- WithShifts_instrs.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

CPUID 方式での命令セットの説明

次のサブセクションでは、CPUID 方式における命令セットの記述方法と、命令を基本命令または非基本命令に分類する方法について説明します。

- *命令分類の一般的な規則*
- *データ処理命令*
- *乗算命令* P. B5-21
- *分岐* P. B5-21
- *単一ワードロードまたはストア命令* P. B5-21
- *複数ワードロードまたはストア命令* P. B5-21
- *PSR の Q フラグのサポート* P. B5-21

命令分類の一般的な規則

2 つの一般的な規則が、このセクションでの命令分類の説明に適用されます。

1. 命令が基本命令であることを示す規則は、その命令が特定の命令セットで使用可能なことを保証するわけではありません。例えば、このセクションで示される規則では、`MOV R0, #123456789` が基本命令として分類されますが、この命令は既存のすべての ARM 命令セットで使用できるわけではありません。

2. 命令が条件付きまたは無条件であるかどうかと、命令が基本命令かどうかは無関係です。

データ処理命令

データ処理命令は、次に示すものです。

```
ADC   ADD   AND   ASR   BIC   CMN   CMP   EOR   LSL   LSR   MOV   MVN
NEG   ORN   ORR   ROR   RRX   RSB   RSC   SBC   SUB   TEQ   TST
```

このグループの命令は、次の両方の条件に該当する場合は基本命令です。

- 2 つ目のソースオペランド、または `MOV` 命令か `MVN` 命令の唯一のソースオペランドが、イミディエートまたはシフトされていないレジスタである。

  注

  ソースオペランドがシフトしたレジスタである `MOV` 命令は、等価の `ASR`、`LSL`、`LSR`、`ROR`、または `RRX` 命令として扱う必要があります。*MOV (シフトしたレジスタ)* P. A8-198 を参照して下さい。

- 命令が、*SUBS PC*、*LR*、*および関連する命令* P. B6-25 で説明されている例外からの復帰命令の 1 つではない。

これらの条件のいずれかが該当しない場合、その命令は非基本命令です。命令セット属性レジスタに含まれる、次の属性のいずれかまたは両方が、非基本データ処理命令のサポートを示します。

- PSR_instrs
- WithShifts_instrs

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Multiply instructions

The classification of multiply instructions is:

- `MUL` instructions are always basic instructions
- all other multiply instructions, and all multiply-accumulate instructions, are non-basic instructions.

Branches

All `B` and `BL` instructions are basic instructions.

Load or Store single word instructions

The instructions in this group are:

`LDR` `LDRB` `LDRH` `LDRSB` `LDRSH` `STR` `STRB` `STRH`

An instruction in this group is a basic instruction if its addressing mode is one of these forms:

- `[Rn, #immediate]`
- `[Rn, #-immediate]`
- `[Rn, Rm]`
- `[Rn, -Rm]`.

A Load or Store single word instruction with any other addressing mode is a non-basic instruction. One or more of these attributes in the Instruction Set Attribute registers shows the support for these instructions:

- WithShifts_instrs
- Writeback_instrs
- Unpriv_instrs.

Load or Store multiple word instructions

The Load or Store multiple word instructions are:

`LDM<mode>` `STM<mode>` `PUSH` `POP`

A limited number of variants of these instructions are non-basic. The Except_instrs attribute in the Instruction Set Attribute registers shows the support for these instructions. For details of these non-basic instructions see *c0, Instruction Set Attribute Register 1 (ID_ISAR1)* on page B5-25.

All other forms of these instructions are always basic instructions.

Q flag support in the PSRs

The Q flag is present in the CPSR and SPSRs when one or more of these conditions apply to the Instruction Set Attribute register attributes:

- MultS_instrs ≥ 2
- Saturate_instrs ≥ 1
- SIMD_instrs ≥ 1.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

乗算命令

乗算命令の分類は次のとおりです。

- `MUL` 命令は常に基本命令です。
- 他のすべての乗算命令と、すべての積和命令は非基本命令です。

分岐

すべての `B` 命令と `BL` 命令は基本命令です。

単一ワードロードまたはストア命令

このグループの命令は、次に示すものです。

`LDR` `LDRB` `LDRH` `LDRSB` `LDRSH` `STR` `STRB` `STRH`

このグループの命令は、そのアドレシングモードが次のいずれかの形式である場合に基本命令です。

- `[Rn, #immediate]`
- `[Rn, #-immediate]`
- `[Rn, Rm]`
- `[Rn, -Rm]`

他のアドレシングモードを持つ、単一ワードロードまたはストア命令は、すべて非基本命令です。命令セット属性レジスタで、次に示す属性のうち 1 つ以上が、これらの命令のサポートを示します。

- WithShifts_instrs
- Writeback_instrs
- Unpriv_instrs

複数ワードロードまたはストア命令

複数ワードロードまたはストア命令は、次に示すものです。

`LDM<mode>` `STM<mode>` `PUSH` `POP`

これらの命令の一部のバリアントは非基本命令です。命令セット属性レジスタ内の Except_instrs 属性が、これらの命令のサポートを示します。これらの非基本命令の詳細については、*c0、命令セット属性レジスタ 1 (ID_ISAR1)* P. B5-25 を参照して下さい。

これらの命令の、他のすべての形式は常に基本命令です。

PSR の Q フラグのサポート

命令セット属性レジスタの属性に、次の条件のうち 1 つ以上が該当する場合、Q フラグが CPSR と SPSR に存在します。

- MultS_instrs ≥ 2
- Saturate_instrs ≥ 1
- SIMD_instrs ≥ 1

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Summary of Instruction Set Attribute register attributes

The Instruction Set Attribute registers use a set of attributes to indicate the non-basic instructions supported by the processor. The descriptions of the non-basic instructions in *Instruction set descriptions in the CPUID scheme* on page B5-20 include the attribute or attributes used to indicate support for each category of non-basic instructions. Table B5-1 lists all of these attributes in alphabetical order, and shows which Instruction Set Attribute register holds each attribute.

Table B5-1 Alphabetic list of Instruction Set Attribute registers attributes

Attribute	Register
Barrier_instrs	*c0, Instruction Set Attribute Register 4 (ID_ISAR4)* on page B5-31
BitCount_instrs	*c0, Instruction Set Attribute Register 0 (ID_ISAR0)* on page B5-24
Bitfield_instrs	*c0, Instruction Set Attribute Register 0 (ID_ISAR0)* on page B5-24
CmpBranch_instrs	*c0, Instruction Set Attribute Register 0 (ID_ISAR0)* on page B5-24
Coproc_instrs	*c0, Instruction Set Attribute Register 0 (ID_ISAR0)* on page B5-24
Debug_instrs	*c0, Instruction Set Attribute Register 0 (ID_ISAR0)* on page B5-24
Divide_instrs	*c0, Instruction Set Attribute Register 0 (ID_ISAR0)* on page B5-24
Endian_instrs	*c0, Instruction Set Attribute Register 1 (ID_ISAR1)* on page B5-25
Except_AR_instrs	*c0, Instruction Set Attribute Register 1 (ID_ISAR1)* on page B5-25
Except_instrs	*c0, Instruction Set Attribute Register 1 (ID_ISAR1)* on page B5-25
Extend_instrs	*c0, Instruction Set Attribute Register 1 (ID_ISAR1)* on page B5-25
IfThen_instrs	*c0, Instruction Set Attribute Register 1 (ID_ISAR1)* on page B5-25
Immediate_instrs	*c0, Instruction Set Attribute Register 1 (ID_ISAR1)* on page B5-25
Interwork_instrs	*c0, Instruction Set Attribute Register 1 (ID_ISAR1)* on page B5-25
Jazelle_instrs	*c0, Instruction Set Attribute Register 1 (ID_ISAR1)* on page B5-25
LoadStore_instrs	*c0, Instruction Set Attribute Register 2 (ID_ISAR2)* on page B5-27
MemHint_instrs	*c0, Instruction Set Attribute Register 2 (ID_ISAR2)* on page B5-27
Mult_instrs	*c0, Instruction Set Attribute Register 2 (ID_ISAR2)* on page B5-27
MultiAccessInt_instrs	*c0, Instruction Set Attribute Register 2 (ID_ISAR2)* on page B5-27
MultS_instrs	*c0, Instruction Set Attribute Register 2 (ID_ISAR2)* on page B5-27
MultU_instrs	*c0, Instruction Set Attribute Register 2 (ID_ISAR2)* on page B5-27

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

命令セット属性レジスタの属性の要約

命令セット属性レジスタは、プロセッサでサポートされている非基本命令を示すため、一連の属性を使用します。*CPUID 方式での命令セットの説明* P. B5-20 における非基本命令の説明には、非基本命令の各カテゴリのサポートを示すために使用される属性が含まれています。これらの属性のアルファベット順のリストと、各属性を保持する命令セット属性レジスタを、表 B5-1 に示します。

表 B5-1 命令セット属性レジスタの属性のアルファベット順のリスト

属性	レジスタ
Barrier_instrs	*c0、命令セット属性レジスタ 4 (ID_ISAR4)* P. B5-31
BitCount_instrs	*c0、命令セット属性レジスタ 0 (ID_ISAR0)* P. B5-24
Bitfield_instrs	*c0、命令セット属性レジスタ 0 (ID_ISAR0)* P. B5-24
CmpBranch_instrs	*c0、命令セット属性レジスタ 0 (ID_ISAR0)* P. B5-24
Coproc_instrs	*c0、命令セット属性レジスタ 0 (ID_ISAR0)* P. B5-24
Debug_instrs	*c0、命令セット属性レジスタ 0 (ID_ISAR0)* P. B5-24
Divide_instrs	*c0、命令セット属性レジスタ 0 (ID_ISAR0)* P. B5-24
Endian_instrs	*c0、命令セット属性レジスタ 1 (ID_ISAR1)* P. B5-25
Except_AR_instrs	*c0、命令セット属性レジスタ 1 (ID_ISAR1)* P. B5-25
Except_instrs	*c0、命令セット属性レジスタ 1 (ID_ISAR1)* P. B5-25
Extend_instrs	*c0、命令セット属性レジスタ 1 (ID_ISAR1)* P. B5-25
IfThen_instrs	*c0、命令セット属性レジスタ 1 (ID_ISAR1)* P. B5-25
Immediate_instrs	*c0、命令セット属性レジスタ 1 (ID_ISAR1)* P. B5-25
Interwork_instrs	*c0、命令セット属性レジスタ 1 (ID_ISAR1)* P. B5-25
Jazelle_instrs	*c0、命令セット属性レジスタ 1 (ID_ISAR1)* P. B5-25
LoadStore_instrs	*c0、命令セット属性レジスタ 2 (ID_ISAR2)* P. B5-27
MemHint_instrs	*c0、命令セット属性レジスタ 2 (ID_ISAR2)* P. B5-27
Mult_instrs	*c0、命令セット属性レジスタ 2 (ID_ISAR2)* P. B5-27
MultiAccessInt_instrs	*c0、命令セット属性レジスタ 2 (ID_ISAR2)* P. B5-27
MultS_instrs	*c0、命令セット属性レジスタ 2 (ID_ISAR2)* P. B5-27
MultU_instrs	*c0、命令セット属性レジスタ 2 (ID_ISAR2)* P. B5-27

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table B5-1 Alphabetic list of Instruction Set Attribute registers attributes (continued)

Attribute	Register
PSR_AR_instrs	*c0, Instruction Set Attribute Register 2 (ID_ISAR2)* on page B5-27
PSR_M_instrs	*c0, Instruction Set Attribute Register 4 (ID_ISAR4)* on page B5-31
Reversal_instrs	*c0, Instruction Set Attribute Register 2 (ID_ISAR2)* on page B5-27
Saturate_instrs	*c0, Instruction Set Attribute Register 3 (ID_ISAR3)* on page B5-29
SIMD_instrs	*c0, Instruction Set Attribute Register 3 (ID_ISAR3)* on page B5-29
SMC_instrs	*c0, Instruction Set Attribute Register 4 (ID_ISAR4)* on page B5-31
SVC_instrs	*c0, Instruction Set Attribute Register 3 (ID_ISAR3)* on page B5-29
Swap_instrs	*c0, Instruction Set Attribute Register 0 (ID_ISAR0)* on page B5-24
SynchPrim_instrs	*c0, Instruction Set Attribute Register 3 (ID_ISAR3)* on page B5-29
SynchPrim_instrs_frac	*c0, Instruction Set Attribute Register 4 (ID_ISAR4)* on page B5-31
TabBranch_instrs	*c0, Instruction Set Attribute Register 3 (ID_ISAR3)* on page B5-29
ThumbCopy_instrs	*c0, Instruction Set Attribute Register 3 (ID_ISAR3)* on page B5-29
ThumbEE_extn_instrs	*c0, Instruction Set Attribute Register 3 (ID_ISAR3)* on page B5-29
TrueNOP_instrs	*c0, Instruction Set Attribute Register 3 (ID_ISAR3)* on page B5-29
Unpriv_instrs	*c0, Instruction Set Attribute Register 4 (ID_ISAR4)* on page B5-31
WithShifts_instrs	*c0, Instruction Set Attribute Register 4 (ID_ISAR4)* on page B5-31
Writeback_instrs	*c0, Instruction Set Attribute Register 4 (ID_ISAR4)* on page B5-31

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 B5-1 命令セット属性レジスタの属性のアルファベット順のリスト（続き）

属性	レジスタ
PSR_AR_instrs	*c0、命令セット属性レジスタ 2 (ID_ISAR2)* P. B5-27
PSR_M_instrs	*c0、命令セット属性レジスタ 4 (ID_ISAR4)* P. B5-31
Reversal_instrs	*c0、命令セット属性レジスタ 2 (ID_ISAR2)* P. B5-27
Saturate_instrs	*c0、命令セット属性レジスタ 3 (ID_ISAR3)* P. B5-29
SIMD_instrs	*c0、命令セット属性レジスタ 3 (ID_ISAR3)* P. B5-29
SMC_instrs	*c0、命令セット属性レジスタ 4 (ID_ISAR4)* P. B5-31
SVC_instrs	*c0、命令セット属性レジスタ 3 (ID_ISAR3)* P. B5-29
Swap_instrs	*c0、命令セット属性レジスタ 0 (ID_ISAR0)* P. B5-24
SynchPrim_instrs	*c0、命令セット属性レジスタ 3 (ID_ISAR3)* P. B5-29
SynchPrim_instrs_frac	*c0、命令セット属性レジスタ 4 (ID_ISAR4)* P. B5-31
TabBranch_instrs	*c0、命令セット属性レジスタ 3 (ID_ISAR3)* P. B5-29
ThumbCopy_instrs	*c0、命令セット属性レジスタ 3 (ID_ISAR3)* P. B5-29
ThumbEE_extn_instrs	*c0、命令セット属性レジスタ 3 (ID_ISAR3)* P. B5-29
TrueNOP_instrs	*c0、命令セット属性レジスタ 3 (ID_ISAR3)* P. B5-29
Unpriv_instrs	*c0、命令セット属性レジスタ 4 (ID_ISAR4)* P. B5-31
WithShifts_instrs	*c0、命令セット属性レジスタ 4 (ID_ISAR4)* P. B5-31
Writeback_instrs	*c0、命令セット属性レジスタ 4 (ID_ISAR4)* P. B5-31

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c0, Instruction Set Attribute Register 0 (ID_ISAR0)

The format of the ID_ISAR0 is:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
Reserved, RAZ	Divide _instrs	Debug _instrs	Coproc _instrs	CmpBranch _instrs	Bitfield _instrs	BitCount _instrs	Swap _instrs

Bits [31:28] Reserved, RAZ.

Divide_instrs, bits [27:24]

Indicates the supported Divide instructions. Permitted values are:

0b0000 . None supported.

0b0001 . Adds support for `SDIV` and `UDIV`.

Debug_instrs, bits [23:20]

Indicates the supported Debug instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for `BKPT`.

Coproc_instrs, bits [19:16]

Indicates the supported Coprocessor instructions. Permitted values are:

0b0000 None supported, except for separately attributed architectures including CP15, CP14, and Advanced SIMD and VFP.

0b0001 Adds support for generic `CDP`, `LDC`, `MCR`, `MRC`, and `STC`.

0b0010 As for 0b0001, and adds generic `CDP2`, `LDC2`, `MCR2`, `MRC2`, and `STC2`.

0b0011 As for 0b0010, and adds generic `MCRR` and `MRRC`.

0b0100 As for 0b0011, and adds generic `MCRR2` and `MRRC2`.

CmpBranch_instrs, bits [15:12]

Indicates the supported combined Compare and Branch instructions in the Thumb instruction set. Permitted values are:

0b0000 None supported.

0b0001 Adds support for `CBNZ` and `CBZ`.

Bitfield_instrs, bits [11:8]

Indicates the supported BitField instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for `BFC`, `BFI`, `SBFX`, and `UBFX`.

BitCount_instrs, bits [7:4]

Indicates the supported Bit Counting instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for `CLZ`.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

c0、命令セット属性レジスタ 0 (ID_ISAR0)

ID_ISAR0 のフォーマットは次のとおりです。

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
予約、RAZ	Divide _instrs	Debug _instrs	Coproc _instrs	CmpBranch _instrs	Bitfield _instrs	BitCount _instrs	Swap _instrs

ビット [31:28] 予約、RAZ

Divide_instrs、ビット [27:24]

サポートされている除算命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 `SDIV` と `UDIV` のサポートが追加されています。

Debug_instrs、ビット [23:20]

サポートされているデバッグ命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 `BKPT` のサポートが追加されています。

Coproc_instrs、ビット [19:16]

サポートされているコプロセッサ命令を示します。許される値は次のとおりです。

0b0000 CP15、CP14、アドバンスト SIMD および VFP など、独立の属性が指定されるアーキテクチャを除いて、サポートされていません。

0b0001 汎用の `CDP`、`LDC`、`MCR`、`MRC`、`STC` のサポートが追加されています。

0b0010 0b0001 の命令に加えて、汎用の `CDP2`、`LDC2`、`MCR2`、`MRC2`、`STC2` が追加されています。

0b0011 0b0010 の命令に加えて、汎用の `MCRR` と `MRRC` が追加されています。

0b0100 0b0011 の命令に加えて、汎用の `MCRR2` と `MRRC2` が追加されています。

CmpBranch_instrs、ビット [15:12]

サポートされている、Thumb 命令セットの比較命令と分岐命令の組み合わせを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 `CBNZ` と `CBZ` のサポートが追加されています。

Bitfield_instrs、ビット [11:8]

サポートされているビットフィールド命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 `BFC`、`BFI`、`SBFX`、`UBFX` のサポートが追加されています。

BitCount_instrs、ビット [7:4]

サポートされているビットカウント命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 `CLZ` のサポートが追加されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Swap_instrs, bits [3:0]

Indicates the supported Swap instructions in the ARM instruction set. Permitted values are:

0b0000 None supported.

0b0001 Adds support for SWP and SWPB.

c0, Instruction Set Attribute Register 1 (ID_ISAR1)

The format of the IID_ISAR1 is:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
Jazelle _instrs	Interwork _instrs	Immediate _instrs	IfThen _instrs	Extend _instrs	Except_AR _instrs	Except _instrs	Endian _instrs

Jazelle_instrs, bits [31:28]

Indicates the supported Jazelle extension instructions. Permitted values are:

0b0000 No support for Jazelle.

0b0001 Adds support for BXJ instruction, and the J bit in the PSR.
This setting might indicate a trivial implementation of Jazelle support.

Interwork_instrs, bits [27:24]

Indicates the supported Interworking instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for BX instruction, and the T bit in the PSR.

0b0010 As for 0b0001, and adds support for BLX instruction. PC loads have BX-like behavior.

0b0011 As for 0b0010, but guarantees that data-processing instructions in the ARM instruction set with the PC as the destination and the S bit clear have BX-like behavior.

Note

A value of 0b0000, 0b0001, or 0b0010 in this field does not guarantee that an ARM data-processing instruction with the PC as the destination and the S bit clear behaves like an old MOV PC instruction, ignoring bits [1:0] of the result. With these values of this field:

- if bits [1:0] of the result value are 0b00 then the processor remains in ARM state
- if bits [1:0] are 0b01, 0b10 or 0b11, the result must be treated as UNPREDICTABLE.

Immediate_instrs, bits [23:20]

Indicates the support for data-processing instructions with long immediates. Permitted values are:

0b0000 None supported.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

Swap_instrs、ビット [3:0]

サポートされている、ARM 命令セットのスワップ命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 SWP と SWPB のサポートが追加されています。

c0、命令セット属性レジスタ 1 (ID_ISAR1)

ID_ISAR1 のフォーマットは次のとおりです。

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
Jazelle _instrs	Interwork _instrs	Immediate _instrs	IfThen _instrs	Extend _instrs	Except_AR _instrs	Except _instrs	Endian _instrs

Jazelle_instrs、ビット [31:28]

サポートされている Jazelle 拡張機能命令を示します。許される値は次のとおりです。

0b0000 Jazelle がサポートされていません。

0b0001 BXJ 命令と、PSR の J ビットのサポートが追加されています。
この設定は、Jazelle のトリビアル実装のサポートを示す場合があります。

Interwork_instrs、ビット [27:24]

サポートされているインターワーキング命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 BX 命令と、PSR の T ビットのサポートが追加されています。

0b0010 0b0001 のものに加えて、BLX 命令のサポートが追加されています。PC ロードは、BX と同様の動作を行います。

0b0011 0b0010 の場合と同様ですが、デスティネーションとして PC を含む ARM 命令セットでのデータ処理命令、および S ビットのクリアは、BX と同様の動作を行います。

注

このフィールドの値が 0b0000、0b0001、または 0b0010 である場合、デスティネーションとして PC を含む ARM データ処理命令、および S ビットのクリアが、従来の MOV PC 命令と同様の動作を行い、結果のビット [1:0] が無視される保証はありません。このフィールドにこれらの値を指定した場合、次のようになります。

- 結果値のビット [1:0] が 0b00 の場合、プロセッサは ARM 状態のままです。
- ビット [1:0] が 0b01、0b10、または 0b11 の場合、結果を予測不能として扱う必要があります。

Immediate_instrs、ビット [23:20]

ロングイミディエートを使用するデータ処理命令のサポートを示します。許される値は次のとおりです。

0b0000 サポートされていません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b0001 Adds support for:

- the MOVT instruction
- the MOV instruction encodings with zero-extended 16-bit immediates
- the Thumb ADD and SUB instruction encodings with zero-extended 12-bit immediates, and the other ADD, ADR and SUB encodings cross-referenced by the pseudocode for those encodings.

IfThen_instrs, bits [19:16]

Indicates the supported IfThen instructions in the Thumb instruction set. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the IT instructions, and for the IT bits in the PSRs.

Extend_instrs, bits [15:12]

Indicates the supported Extend instructions. Permitted values are:

0b0000 No scalar sign-extend or zero-extend instructions are supported, where scalar instructions means non-Advanced SIMD instructions.

0b0001 Adds support for the SXTB, SXTH, UXTB, and UXTH instructions.

0b0010 As for 0b0001, and adds support for the SXTB16, SXTAB, SXTAB16, SXTAH, UXTB16, UXTAB, UXTAB16, and UXTAH instructions.

Note

In addition:

- the shift options on these instructions are available only if the WithShifts_instrs attribute is 0b0011 or greater
- the SXTAB16, SXTB16, UXTAB16, and UXTB16 instructions are available only if both:
  - the Extend_instrs attribute is 0b0010 or greater
  - the SIMD_instrs attribute is 0b0011 or greater.

Except_AR_instrs, bits [11:8]

Indicates the supported A and R profile exception-handling instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the SRS and RFE instructions, and the A and R profile forms of the CPS instruction.

Except_instrs, bits [7:4]

Indicates the supported exception-handling instructions in the ARM instruction set. Permitted values are:

0b0000 Not supported. This indicates that the User bank and Exception return forms of the LDM and STM instructions are not supported.

0b0001 Adds support for the LDM (exception return), LDM (user registers) and STM (user registers) instruction versions.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

0b0001 次のサポートが追加されています。

- MOVT 命令
- ゼロ拡張された 16 ビットイミディエートを使用する、MOV 命令のエンコード
- ゼロ拡張された 12 ビットイミディエートを使用する Thumb ADD 命令および SUB 命令のエンコードと、これらのエンコードの擬似コードで相互参照される、他の ADD、ADR、SUB のエンコード

IfThen_instrs、ビット [19:16]

サポートされている、Thumb 命令セットの IfThen 命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 IT 命令と、PSR の IT ビットのサポートが追加されています。

Extend_instrs、ビット [15:12]

サポートされている拡張命令を示します。許される値は次のとおりです。

0b0000 スカラ符号拡張またはゼロ拡張命令がサポートされていません。スカラ命令は非アドバンスト SIMD 命令を意味します。

0b0001 SXTB、SXTH、UXTB、UXTH 命令のサポートが追加されています。

0b0010 0b0001 の命令に加えて、SXTB16、SXTAB、SXTAB16、SXTAH、UXTB16、UXTAB、UXTAB16、UXTAH 命令のサポートが追加されています。

注

さらに、次の規則が適用されます。

- WithShifts_instrs 属性が 0b0011 以上の場合にのみ、これらの命令についてシフトオプションが使用可能です。
- 次の両方が満たされる場合のみ、SXTAB16、SXTB16、UXTAB16、UXTB16 命令が使用可能です。
  - Extend_instrs 属性が 0b0010 以上である
  - SIMD_instrs 属性が 0b0011 以上である

Except_AR_instrs、ビット [11:8]

サポートされている、A および R プロファイルの例外処理命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 SRS 命令と RFE 命令と、A および R プロファイル形式の CPS 命令のサポートが追加されています。

Except_instrs、ビット [7:4]

サポートされている、ARM 命令セットの例外処理命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。これは、ユーザバンクおよび例外からの復帰形式の、LDM 命令と STM 命令がサポートされていないことを意味します。

0b0001 LDM（例外からの復帰）、LDM（ユーザレジスタ）、STM（ユーザレジスタ）命令バージョンのサポートが追加されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Endian_instrs, bits [3:0]

Indicates the supported Endian instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the `SETEND` instruction, and the E bit in the PSRs.

c0, Instruction Set Attribute Register 2 (ID_ISAR2)

The format of the ID_ISAR2 is:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
Reversal _instrs	PSR_AR _instrs	MultU _instrs	MultS _instrs	Mult _instrs	MultiAccess Int_instrs	MemHint _instrs	LoadStore _instrs

Reversal_instrs, bits [31:28]

Indicates the supported Reversal instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the `REV`, `REV16`, and `REVSH` instructions.

0b0010 As for 0b0001, and adds support for the `RBIT` instruction.

PSR_AR_instrs, bits [27:24]

Indicates the supported A and R profile instructions to manipulate the PSR. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the `MRS` and `MSR` instructions, and the exception return forms of data-processing instructions described in *SUBS PC, LR and related instructions* on page B6-25.

Note

The exception return forms of the data-processing instructions are:

- In the ARM instruction set, data-processing instructions with the PC as the destination and the S bit set. These instructions might be affected by the WithShifts attribute.
- In the Thumb instruction set, the `SUBS PC,LR,#N` instruction.

MultU_instrs, bits [23:20]

Indicates the supported advanced unsigned Multiply instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the `UMULL` and `UMLAL` instructions.

0b0010 As for 0b0001, and adds support for the `UMAAL` instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Endian_instrs、ビット [3:0]

サポートされているエンディアン命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 `SETEND` 命令と、PSR の E ビットのサポートが追加されています。

c0、命令セット属性レジスタ 2 (ID_ISAR2)

ID_ISAR2 のフォーマットは次のとおりです。

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
Reversal _instrs	PSR_AR _instrs	MultU _instrs	MultS _instrs	Mult _instrs	MultiAccess Int_instrs	MemHint _instrs	LoadStore _instrs

Reversal_instrs、ビット [31:28]

サポートされている反転命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 `REV`、`REV16`、`REVSH` 命令のサポートが追加されています。

0b0010 0b0001 の命令に加えて、`RBIT` 命令のサポートが追加されています。

PSR_AR_instrs、ビット [27:24]

サポートされている、PSR を操作するための A および R プロファイル命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 `MRS` 命令と `MSR` 命令、および *SUBS PC、LR、および関連する命令* P. B6-25 で説明されている、例外からの復帰形式のデータ処理命令のサポートが追加されています。

注

例外からの復帰形式のデータ処理命令の特徴は次のとおりです。

- ARM 命令セットの、デスティネーションとして PC を含み、S ビットがセットされたデータ処理命令。これらの命令は、WithShifts 属性の影響を受ける可能性があります。
- Thumb 命令セットの `SUBS PC,LR,#N` 命令

MultU_instrs、ビット [23:20]

サポートされている、高度な符号なし乗算命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 `UMULL` 命令と `UMLAL` 命令のサポートが追加されています。

0b0010 0b0001 の命令に加えて、`UMAAL` 命令のサポートが追加されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

MultS_instrs, bits [19:16]

Indicates the supported advanced signed Multiply instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the SMULL and SMLAL instructions.

0b0010 As for 0b0001, and adds support for the SMLABB, SMLABT, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, and SMULWT instructions.

Also adds support for the Q bit in the PSRs.

0b0011 As for 0b0010, and adds support for the SMLAD, SMLADX, SMLALD, SMLALDX, SMLSD, SMLSDX, SMLSLD, SMLSLDX, SMMLA, SMMLAR, SMMLS, SMMLSR, SMMUL, SMMULR, SMUAD, SMUADX, SMUSD, and SMUSDX instructions.

Mult_instrs, bits [15:12]

Indicates the supported additional Multiply instructions. Permitted values are:

0b0000 No additional instructions supported. This means only MUL is supported.

0b0001 Adds support for the MLA instruction.

0b0010 As for 0b0001, and adds support for the MLS instruction.

MultiAccessInt_instrs, bits [11:8]

Indicates the support for multi-access interruptible instructions. Permitted values are:

0b0000 None supported. This means the LDM and STM instructions are not interruptible.

0b0001 LDM and STM instructions are restartable.

0b0010 LDM and STM instructions are continuable.

MemHint_instrs, bits [7:4]

Indicates the supported Memory Hint instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the PLD instruction.

0b0010 Adds support for the PLD instruction.

In the MemHint_instrs field, entries of 0b0001 and 0b0010 have identical meanings.

0b0011 As for 0b0001 (or 0b0010), and adds support for the PLI instruction.

0b0100 As for 0b0011, and adds support for the PLDW instruction.

LoadStore_instrs, bits [3:0]

Indicates the supported additional load/store instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the LDRD and STRD instructions.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

MultS_instrs、ビット [19:16]

サポートされている、高度な符号付き乗算命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 SMULL 命令と SMLAL 命令のサポートが追加されています。

0b0010 0b0001 の命令に加えて、SMLABB、SMLABT、SMLALBB、SMLALBT、SMLALTB、SMLALTT、SMLATB、SMLATT、SMLAWB、SMLAWT、SMULBB、SMULBT、SMULTB、SMULTT、SMULWB、SMULWT 命令のサポートが追加されています。

PSR の Q ビットのサポートも追加されています。

0b0011 0b0010 の命令に加えて、SMLAD、SMLADX、SMLALD、SMLALDX、SMLSD、SMLSDX、SMLSLD、SMLSLDX、SMMLA、SMMLAR、SMMLS、SMMLSR、SMMUL、SMMULR、SMUAD、SMUADX、SMUSD、SMUSDX 命令のサポートが追加されています。

Mult_instrs、ビット [15:12]

サポートされている、他の乗算命令を示します。許される値は次のとおりです。

0b0000 他の命令はサポートされていません。これは、MUL のみがサポートされていることを意味します。

0b0001 MLA 命令のサポートが追加されています。

0b0010 0b0001 の命令に加えて、MLS 命令のサポートが追加されています。

MultiAccessInt_instrs、ビット [11:8]

マルチアクセス割り込み可能命令のサポートを示します。許される値は次のとおりです。

0b0000 サポートされていません。これは、LDM 命令と STM 命令が割り込み可能でないことを意味します。

0b0001 LDM 命令と STM 命令が再起動可能です。

0b0010 LDM 命令と STM 命令が続行可能です。

MemHint_instrs、ビット [7:4]

サポートされているメモリヒント命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 PLD 命令のサポートが追加されています。

0b0010 PLD 命令のサポートが追加されています。

MemHint_instrs フィールドでは、エントリの b0001 と 0b0010 は同じ意味です。

0b0011 0b0001（または 0b0010）の命令に加えて、PLI 命令のサポートが追加されています。

0b0100 0b0011 の命令に加えて、PLDW 命令のサポートが追加されています。

LoadStore_instrs、ビット [3:0]

サポートされている、追加のロード / ストア命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 LDRD 命令と STRD 命令のサポートが追加されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c0, Instruction Set Attribute Register 3 (ID_ISAR3)

The format of the ID_ISAR3 is:

31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
ThumbEE_ extn_instrs	TrueNOP _instrs	ThumbCopy _instrs	TabBranch _instrs	SynchPrim _instrs	SVC _instrs	SIMD _instrs	Saturate _instrs

ThumbEE_extn_instrs, bits [31:28]

Indicates the supported Thumb Execution Environment (ThumbEE) extension instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the `ENTERX` and `LEAVEX` instructions, and modifies the load behavior to include null checking.

Note

This field can only have a value other than 0b0000 when the PFR0 register State3 field has a value of 0b0001, see *c0, Processor Feature Register 0 (ID_PFR0)* on page B5-4.

TrueNOP_instrs, bits [27:24]

Indicates the support for True NOP instructions. Permitted values are:

0b0000 None supported. This means there are no `NOP` instructions that do not have any register dependencies.

0b0001 Adds true `NOP` instructions in both the Thumb and ARM instruction sets. Also permits additional NOP-compatible hints.

ThumbCopy_instrs, bits [23:20]

Indicates the supported Thumb non flag-setting `MOV` instructions. Permitted values are:

0b0000 Not supported. This means that in the Thumb instruction set, encoding T1 of the `MOV` (register) instruction does not support a copy from a low register to a low register.

0b0001 Adds support for Thumb instruction set encoding T1 of the `MOV` (register) instruction, copying from a low register to a low register.

TabBranch_instrs, bits [19:16]

Indicates the supported Table Branch instructions in the Thumb instruction set. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the `TBB` and `TBH` instructions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c0、命令セット属性レジスタ 3 (ID_ISAR3)

ID_ISAR3 のフォーマットは次のとおりです。

31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
ThumbEE_ extn_instrs	TrueNOP _instrs	ThumbCopy _instrs	TabBranch _instrs	SynchPrim _instrs	SVC _instrs	SIMD _instrs	Saturate _instrs

ThumbEE_extn_instrs、ビット [31:28]

サポートされている、Thumb 実行環境 (ThumbEE) 拡張命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 `ENTERX` 命令と `LEAVEX` 命令のサポートが追加され、NULL チェックを含めるようにロード動作が変更されています。

注

PFR0 レジスタの State3 フィールドの値が 0b0001 の場合は、このフィールドに 0b0000 の値を設定できません。*c0、プロセッサ機能レジスタ 0 (ID_PFR0)* P. B5-4 を参照して下さい。

TrueNOP_instrs、ビット [27:24]

真の NOP 命令のサポートを示します。許される値は次のとおりです。

0b0000 サポートされていません。これは、レジスタに依存しない `NOP` 命令が存在しないことを意味します。

0b0001 Thumb 命令セットと ARM 命令セットの両方に、真の `NOP` 命令が追加されています。追加の NOP 互換ヒントも許可されます。

ThumbCopy_instrs、ビット [23:20]

サポートされている、Thumb のフラグをセットしない `MOV` 命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。これは、Thumb 命令セットの `MOV`（レジスタ）命令のエンコード T1 で、下位レジスタから下位レジスタへのコピーがサポートされていないことを意味します。

0b0001 下位レジスタから下位レジスタにコピーする、Thumb 命令セットの `MOV`（レジスタ）命令のエンコード T1 のサポートが追加されています。

TabBranch_instrs、ビット [19:16]

サポートされている、Thumb 命令セットのテーブル分岐命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 `TBB` 命令と `TBH` 命令のサポートが追加されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

SynchPrim_instrs, bits [15:12]

This field is used with the SynchPrim_instrs_frac field of ID_ISAR4 to indicate the supported Synchronization Primitive instructions. Table B5-2 shows the permitted values of these fields:

Table B5-2 Synchronization Primitives support

SynchPrim_instrs	SynchPrim_instrs_frac	Supported Synchronization Primitives
0000	0000	None supported
0001	0000	Adds support for the LDREX and STREX instructions.
0001	0011	As for [0001,0000], and adds support for the CLREX, LDREXB, LDREXH, STREXB, and STREXH instructions.
0010	0000	As for [0001,0011], and adds support for the LDREXD and STREXD instructions.

All combinations of SynchPrim_instrs and SynchPrim_instrs_frac not shown in Table B5-2 are reserved.

SVC_instrs, bits [11:8]

Indicates the supported SVC instructions. Permitted values are:

0b0000 Not supported.

0b0001 Adds support for the SVC instruction.

Note

The SVC instruction was called the SWI instruction in previous versions of the ARM architecture.

SIMD_instrs, bits [7:4]

Indicates the supported SIMD instructions. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the SSAT and USAT instructions, and for the Q bit in the PSRs.

0b0011 As for 0b0001, and adds support for the PKHBT, PKHTB, QADD16, QADD8, QASX, QSUB16, QSUB8, QSAX, SADD16, SADD8, SASX, SEL, SHADD16, SHADD8, SHASX, SHSUB16, SHSUB8, SHSAX, SSAT16, SSUB16, SSUB8, SSAX, SXTAB16, SXTB16, UADD16, UADD8, UASX, UHADD16, UHADD8, UHASX, UHSUB16, UHSUB8, UHSAX, UQADD16, UQADD8, UQASX, UQSUB16, UQSUB8, UQSAX, USAD8, USADA8, USAT16, USUB16, USUB8, USAX, UXTAB16, and UXTB16 instructions.

Also adds support for the GE[3:0] bits in the PSRs.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

SynchPrim_instrs、ビット [15:12]

このフィールドは、ID_ISAR4 の SynchPrim_instrs_frac フィールドとともに、サポートされている同期化基本命令を示すために使用されます。これらのフィールドに許される値を、表 B5-2 に示します。

表 B5-2 同期化基本命令のサポート

SynchPrim_instrs	SynchPrim_instrs_frac	サポートされる同期化基本命令
0000	0000	サポートされていません。
0001	0000	LDREX 命令と STREX 命令のサポートが追加されています。
0001	0011	[0001、0000] の命令に加えて、CLREX、LDREXB、LDREXH、STREXB、STREXH 命令のサポートが追加されています。
0010	0000	[0001、0011] の命令に加えて、LDREXD 命令と STREXD 命令のサポートが追加されています。

表 B5-2 に示されていない SynchPrim_instrs と SynchPrim_instrs_frac の組み合わせは、予約されています。

SVC_instrs、ビット [11:8]

サポートされている SVC 命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 SVC 命令のサポートが追加されています。

注

以前のバージョンの ARM アーキテクチャでは、SVC 命令が SWI 命令と呼ばれていました。

SIMD_instrs、ビット [7:4]

サポートされている SIMD 命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 SSAT 命令と USAT 命令、および PSR の Q ビットのサポートが追加されています。

0b0011 0b0001 の命令に加えて、PKHBT、PKHTB、QADD16、QADD8、QASX、QSUB16、QSUB8、QSAX、SADD16、SADD8、SASX、SEL、SHADD16、SHADD8、SHASX、SHSUB16、SHSUB8、SHSAX、SSAT16、SSUB16、SSUB8、SSAX、SXTAB16、SXTB16、UADD16、UADD8、UASX、UHADD16、UHADD8、UHASX、UHSUB16、UHSUB8、UHSAX、UQADD16、UQADD8、UQASX、UQSUB16、UQSUB8、UQSAX、USAD8、USADA8、USAT16、USUB16、USUB8、USAX、UXTAB16、UXTB16 命令のサポートが追加されています。

PSR の GE[3:0] ビットのサポートも追加されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

- in the SIMD_instrs field, the value of 0b0010 is reserved
- the SXTAB16, SXTB16, UXTAB16, and UXTB16 instructions are available only if both:
  - the Extend_instrs attribute is 0b0010 or greater
  - the SIMD_instrs attribute is 0b0011 or greater.

Saturate_instrs, bits [3:0]

Indicates the supported Saturate instructions. Permitted values are:

0b0000 None supported. This means no non-Advanced SIMD saturate instructions are supported.

0b0001 Adds support for the QADD, QDADD, QDSUB, and QSUB instructions, and for the Q bit in the PSRs.

c0, Instruction Set Attribute Register 4 (ID_ISAR4)

The format of the ID_ISAR4 is:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
SWP_frac	PSR_M _instrs	SynchPrim _instrs_frac	Barrier _instrs	SMC _instrs	Writeback _instrs	WithShifts _instrs	Unpriv _instrs

SWP_frac, bits [31:28]

Indicates support for the memory system locking the bus for SWP or SWPB instructions. Permitted values are:

0b0000 SWP or SWPB not supported.

0b0001 SWP or SWPB supported but only in a uniprocessor context. SWP and SWPB do not guarantee whether memory accesses from other masters can come between the load memory access and the store memory access of the SWP or SWPB.

This field is valid only if the Swap_instrs field in ID_ISAR0 is zero.

PSR_M_instrs, bits [27:24]

Indicates the supported M profile instructions to modify the PSRs. Permitted values are:

0b0000 None supported.

0b0001 Adds support for the M profile forms of the CPS, MRS and MSR instructions.

SynchPrim_instrs_frac, bits [23:20]

This field is used with the SynchPrim_instrs field of ID_ISAR3 to indicate the supported Synchronization Primitive instructions. Table B5-2 on page B5-30 shows the permitted values of these fields.

All combinations of SynchPrim_instrs and SynchPrim_instrs_frac not shown in Table B5-2 on page B5-30 are reserved.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

- SIMD_instrs フィールドでは、0b0010 の値が予約されています。
- 次の両方が満たされる場合のみ、SXTAB16、SXTB16、UXTAB16、UXTB16 命令が使用可能です。
  - Extend_instrs 属性が 0b0010 以上である
  - SIMD_instrs 属性が 0b0011 以上である

Saturate_instrs、ビット [3:0]

サポートされている飽和命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。これは、非アドバンスト SIMD 飽和命令がサポートされていないことを意味します。

0b0001 QADD、QDADD、QDSUB、QSUB 命令と、PSR の Q ビットのサポートが追加されています。

c0、命令セット属性レジスタ 4 (ID_ISAR4)

ID_ISAR4 のフォーマットは次のとおりです。

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
SWP_frac	PSR_M _instrs	SynchPrim _instrs_frac	Barrier _instrs	SMC _instrs	Writeback _instrs	WithShifts _instrs	Unpriv _instrs

SWP_frac、ビット [31:28]

SWP 命令または SWPB 命令のバスをロックするメモリシステムのサポートを示します。許される値は次のとおりです。

0b0000 SWP または SWPB がサポートされていません。

0b0001 SWP または SWPB がサポートされていますが、ユニプロセッサコンテキストでのみです。SWP と SWPB では、他のマスタからのメモリアクセスが、SWP または SWPB のロード メモリアクセスとストア メモリアクセスとの間で発生するかどうかは保証されません。

このフィールドは、ID_ISAR0 の Swap_instrs フィールドが 0 の場合のみ有効です。

PSR_M_instrs、ビット [27:24]

サポートされている、PSR を変更するための M プロファイル命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 M プロファイル形式の CPS、MRS、MSR 命令のサポートが追加されています。

SynchPrim_instrs_frac、ビット [23:20]

このフィールドは、ID_ISAR3 の SynchPrim_instrs フィールドとともに、サポートされている同期化基本命令を示すために使用されます。これらのフィールドに許される値を、P. B5-30 の表 B5-2 に示します。

P. B5-30 の表 B5-2 に示されていない SynchPrim_instrs と SynchPrim_instrs_frac の組み合わせは、すべて予約されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Barrier_instrs, bits [19:16]

Indicates the supported Barrier instructions in the ARM and Thumb instruction sets. Permitted values are:

0b0000 None supported. Barrier operations are provided only as CP15 operations.

0b0001 Adds support for the DMB, DSB, and ISB barrier instructions.

If this field is set to a value other than 0b0000 then the L1 unified cache field, bits [23:20], must be set to 0b0000.

SMC_instrs, bits [15:12]

Indicates the supported SMC instructions. Permitted values are:

0b0000 Not supported.

0b0001 Adds support for the SMC instruction.

Note

The SMC instruction was called the SMI instruction in previous versions of the ARM architecture.

Writeback_instrs, bits [11:8]

Indicates the support for Writeback addressing modes. Permitted values are:

0b0000 Basic support. Only the LDM, STM, PUSH, POP, SRS, and RFE instructions support writeback addressing modes. These instructions support all of their writeback addressing modes.

0b0001 Adds support for all of the writeback addressing modes defined in ARMv7.

WithShifts_instrs, bits [7:4]

Indicates the support for instructions with shifts. Permitted values are:

0b0000 Nonzero shifts supported only in MOV and shift instructions.

0b0001 Adds support for shifts of loads and stores over the range LSL 0-3.

0b0011 As for 0b0001, and adds support for other constant shift options, both on load/store and other instructions.

0b0100 As for 0b0011, and adds support for register-controlled shift options.

Note

- In this field, the value of 0b0010 is reserved.
- Additions to the basic support indicated by the 0b0000 field value only apply when the encoding supports them. In particular, in the Thumb instruction set there is no difference between the 0b0011 and 0b0100 levels of support.
- MOV instructions with shift options are treated as ASR, LSL, LSR, ROR or RRX instructions, as described in *Data-processing instructions* on page B5-20.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

Barrier_instrs、ビット [19:16]

サポートされている、ARM 命令セットと Thumb 命令セットのバリア命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。バリア命令は CP15 操作としてのみ提供されます。

0b0001 DMB、DSB、ISB バリア命令のサポートが追加されています。

このフィールドを 0b0000 以外の値に設定した場合は、L1 統一キャッシュフィールド（ビット [23:20]）を 0b0000 に設定する必要があります。

SMC_instrs、ビット [15:12]

サポートされている SMC 命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 SMC 命令のサポートが追加されています。

注

以前のバージョンの ARM アーキテクチャでは、SMC 命令が SMI 命令と呼ばれていました。

Writeback_instrs、ビット [11:8]

ライトバック アドレシングモードのサポートを示します。許される値は次のとおりです。

0b0000 基本サポート。LDM、STM、PUSH、POP、SRS、RFE 命令でのみ、ライトバック アドレシングモードがサポートされます。これらの命令では、すべてのライトバック アドレシングモードがサポートされます。

0b0001 ARMv7 で定義されている、すべてのライトバック アドレシングモードのサポートが追加されています。

WithShifts_instrs、ビット [7:4]

シフト付き命令のサポートを示します。許される値は次のとおりです。

0b0000 MOV 命令とシフト命令でのみ非ゼロシフトがサポートされています。

0b0001 LSL 0 ～ 3 の範囲を超えるロードとストアのシフトのサポートが追加されています。

0b0011 0b0001 のサポートに加えて、ロード / ストア命令、および他の命令の両方で、他の定数シフトオプションのサポートが追加されています。

0b0100 0b0011 のサポートに加えて、レジスタ制御シフトオプションのサポートが追加されています。

注

- このフィールドでは、0b0010 の値が予約されています。
- 0b0000 のフィールド値で示される基本サポートへの拡張は、エンコードでそれらがサポートされている場合にのみ適用されます。特に、Thumb 命令セットでは、0b0011 レベルと 0b0100 レベルのサポートに違いがありません。
- *データ処理命令* P. B5-20 で説明されているように、シフトオプション付きの MOV 命令は、ASR、LSL、LSR、ROR、または RRX 命令として扱われます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Unpriv_instrs, bits [3:0]

Indicates the supported Unprivileged instructions. Permitted values are:

0b0000 None supported. No T variant instructions are implemented.

0b0001 Adds support for LDRBT, LDRT, STRBT, and STRT instructions.

0b0010 As for 0b0001, and adds support for LDRHT, LDRSBT, LDRSHT, and STRHT instructions.

c0, Instruction Set Attribute Register 5 (ID_ISAR5)

The format of the ID_ISAR5 is:

Bits [31:0] Reserved, RAZ.

Accessing the Instruction Set Attribute registers

To access the Instruction Set Attribute Registers you read the CP15 registers with <opc1> set to 0, <CRn> set to c0, <CRm> set to c2, and <opc2> set to:

- 0 for the ID_ISAR0
- 1 for the ID_ISAR1
- 2 for the ID_ISAR2
- 3 for the ID_ISAR3
- 4 for the ID_ISAR4
- 5 for the ID_ISAR5.

For example:

```
MRC p15, 0, <Rt>, c0, c2, 3    ; Read Instruction Set Attribute Register 3
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Unpriv_instrs、ビット [3:0]

サポートされている非特権命令を示します。許される値は次のとおりです。

0b0000 サポートされていません。T バリアント命令は実装されていません。

0b0001 LDRBT、LDRT、STRBT、STRT 命令のサポートが追加されています。

0b0010 0b0001 の命令に加えて、LDRHT、LDRSBT、LDRSHT、STRHT 命令のサポートが追加されています。

c0、命令セット属性レジスタ 5 (ID_ISAR5)

ID_ISAR5 のフォーマットは次のとおりです。

ビット [31:0] 予約、RAZ

命令セット属性レジスタへのアクセス

命令セット属性レジスタにアクセスするには、<opc1> を 0、<CRn> を c0、<CRm> を c2、<opc2> を次の値に設定して、CP15 レジスタを読み出します。

- ID_ISAR0 の場合、0
- ID_ISAR1 の場合、1
- ID_ISAR2 の場合、2
- ID_ISAR3 の場合、3
- ID_ISAR4 の場合、4
- ID_ISAR5 の場合、5

次に例を示します。

```
MRC p15, 0, <Rt>, c0, c2, 3    ; Read Instruction Set Attribute Register 3
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B5.3 Advanced SIMD and VFP feature identification registers

When an implementation includes one or both of the optional Advanced SIMD and VFP extensions, the feature identification registers for the extensions are implemented in a common register block. The extensions reside in the coprocessor space for coprocessors CP10 and CP11, and the registers are accessed using the `VMRS` and `VMSR` instructions. For more information, see *Register map of the Advanced SIMD and VFP extension system registers* on page B1-66.

Table B5-3 lists the feature identification registers for the Advanced SIMD and VFP extensions. These are described in the remainder of this section.

When the Security Extensions are implemented, these registers are Common registers.

Table B5-3 Advanced SIMD and VFP feature identification registers

System register	Name	Description
0b0000	FPSID	See *Floating-point System ID Register (FPSID)*
0b0110	MVFR1	See *Media and VFP Feature Register 1 (MVFR1)* on page B5-38
0b0111	MVFR0	See *Media and VFP Feature Register 0 (MVFR0)* on page B5-36

B5.3.1 Floating-point System ID Register (FPSID)

In ARMv7, the FPSID Register provides top-level information about the floating-point implementation.

Note

- In an ARMv7 implementation that includes one or both of the Advanced SIMD and VFP extensions the Media and VFP Feature registers provide details of the implemented VFP architecture.
- The FPSID can be implemented in a system that provides only software emulation of the ARM floating-point instructions.

The ARMv7 format of the FPSID is:

Implementer, bits [31:24]

Implementer codes are the same as those used for the Main ID Register, see:

- *c0, Main ID Register (MIDR)* on page B3-81, for a VMSA implementation
- *c0, Main ID Register (MIDR)* on page B4-32, for a PMSA implementation.

For an implementation by ARM this field is `0x41`, the ASCII code for A.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B5.3 アドバンスト SIMD および VFP 機能識別レジスタ

実装に、オプションのアドバンスト SIMD 拡張機能と VFP 拡張機能のいずれか、または両方が含まれている場合は、これらの拡張機能の機能識別レジスタが、共通レジスタブロックに実装されます。これらの拡張機能は、コプロセッサの CP10 と CP11 のコプロセッサ空間に存在し、そのレジスタは、`VMRS` 命令と `VMSR` 命令を使用してアクセスされます。詳細については、*アドバンスト SIMD および VFP 拡張機能システムレジスタのレジスタマップ* P. B1-66 を参照して下さい。

アドバンスト SIMD 拡張機能と VFP 拡張機能の機能識別レジスタを、表 B5-3 に示します。このセクションの以下の部分では、これらのレジスタについて説明します。

セキュリティ拡張機能が実装されている場合、これらのレジスタは共通レジスタです。

表 B5-3 アドバンスト SIMD および VFP 機能識別レジスタ

システムレジスタ	名前	説明
0b0000	FPSID	*浮動小数点システム ID レジスタ (FPSID)* を参照
0b0110	MVFR1	*メディアおよび VFP 機能レジスタ 1(MVFR1)* P. B5-38 を参照
0b0111	MVFR0	*メディアおよび VFP 機能レジスタ 0 (MVFR0)* P. B5-36 を参照

B5.3.1 浮動小数点システム ID レジスタ (FPSID)

ARMv7 では、FPSID レジスタによって、浮動小数点実装に関するトップレベルの情報が提供されます。

注

- アドバンスト SIMD 拡張機能と VFP 拡張機能のいずれか、または両方が含まれる ARMv7 実装では、メディアおよび VFP 機能レジスタによって、実装されている VFP アーキテクチャの詳細が提供されます。
- FPSID は、ARM 浮動小数点命令のソフトウェアエミュレーションのみを提供するシステムに実装することができます。

FPSID の ARMv7 フォーマットは次のとおりです。

実装者、ビット [31:24]

実装者コードは、メイン ID レジスタに使用されるものと同じです。次のセクションを参照して下さい。

- *c0、メイン ID レジスタ (MIDR)* P. B3-81（VMSA 実装の場合）
- *c0、メイン ID レジスタ (MIDR)* P. B4-32（PMSA 実装の場合）

ARM による実装では、このフィールドは ASCII コードの A を表す `0x41` です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

SW, bit [23] Software flag. This bit is used to indicate that a system provides only software emulation of the VFP floating-point instructions:

0 The system includes hardware support for VFP floating-point operations.

1 The system provides only software emulation of the VFP floating-point instructions.

Subarchitecture, bits [22:16]

Subarchitecture version number. For an implementation by ARM, permitted values are:

0b0000000

VFPv1 architecture with an IMPLEMENTATION DEFINED subarchitecture.

Not permitted in an ARMv7 implementation.

0b0000001

VFPv2 architecture with Common VFP subarchitecture v1.

Not permitted in an ARMv7 implementation.

0b0000010

VFP architecture v3 or later with Common VFP subarchitecture v2. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.

0b0000011

VFP architecture v3 or later with Null subarchitecture. The entire floating-point implementation is in hardware, and no software support code is required. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.

This value can be used only by an implementation that does not support the trap enable bits in the FPSCR, see *Floating-point Status and Control Register (FPSCR)* on page A2-28.

0b0000100

VFP architecture v3 or later with Common VFP subarchitecture v3. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.

For a subarchitecture designed by ARM the most significant bit of this field, register bit [22], is 0. Values with a most significant bit of 0 that are not listed here are reserved.

When the subarchitecture designer is not ARM, the most significant bit of this field, register bit [22], must be 1. Each implementer must maintain its own list of subarchitectures it has designed, starting at subarchitecture version number `0x40`.

Part number, bits [15:8]

An IMPLEMENTATION DEFINED part number for the floating-point implementation, assigned by the implementer.

Variant, bits [7:4]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different production variants of a single product.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

SW、ビット[23] ソフトウェアフラグ。このビットは、システムで VFP 浮動小数点命令のソフトウェアエミュレーションのみが提供されることを示すために使用されます。

0 システムに、VFP 浮動小数点演算のハードウェアサポートが含まれています。

1 システムで、VFP 浮動小数点命令のソフトウェアエミュレーションのみが提供されます。

サブアーキテクチャ、ビット [22:16]

サブアーキテクチャのバージョン番号。ARM による実装の場合、許される値は次のとおりです。

0b000000

実装定義のサブアーキテクチャを含む、VFPv1 アーキテクチャ。

ARMv7 実装では許されません。

0b0000001

共通 VFP サブアーキテクチャ v1 を含む、VFPv2 アーキテクチャ。

ARMv7 実装では許されません。

0b0000010

共通 VFP サブアーキテクチャ v2 を含む、VFP アーキテクチャ v3 またはそれ以降。VFP アーキテクチャのバージョンは、MVFR0 レジスタと MVFR1 レジスタによって示されます。

0b0000011

NULL サブアーキテクチャを含む、VFP アーキテクチャ v3 または以降。浮動小数点実装全体がハードウェアに存在するため、ソフトウェア サポートコードは必要ありません。VFP アーキテクチャのバージョンは、MVFR0 レジスタと MVFR1 レジスタによって示されます。

この値は、FPSCR のトラップイネーブル ビットをサポートしない実装でのみ、使用できます。*浮動小数点ステータスおよび制御レジスタ (FPSCR)* P. A2- 28 を参照して下さい。

0b0000100

共通 VFP サブアーキテクチャ v3 を含む、VFP アーキテクチャ v3 またはそれ以降。VFP アーキテクチャのバージョンは、MVFR0 レジスタと MVFR1 レジスタによって示されます。

ARM によって設計されたサブアーキテクチャの場合は、このフィールドの最上位ビット（ビット [22]）は 0 です。ここに示されていない、最上位ビットが 0 の値は予約されています。

サブアーキテクチャの設計者が ARM 以外の場合は、このフィールドの最上位ビット（ビット [22]）は 1 にする必要があります。実装者は、サブアーキテクチャのバージョン番号が `0x40` から始まる、独自に設計したサブアーキテクチャのリストを維持する必要があります。

部品番号、ビット [15:8]

実装者によって割り当てられる、浮動小数点実装の実装定義の部品番号

バリアント、ビット [7:4]

実装定義のバリアント番号。一般に、このフィールドは、単一の製品について複数の製品バリアントを区別するために使用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the floating-point implementation.

B5.3.2 Media and VFP Feature registers

The Media and VFP Feature registers describe the features provided by the Advanced SIMD and VFP extensions, when an implementation includes either or both of these extensions. For details of the implementation options for these extensions see *Advanced SIMD and VFP extensions* on page A2-20.

In VFPv2, it is IMPLEMENTATION DEFINED whether the Media and VFP Feature registers are implemented.

Note

Often, the complete implementation of a VFP architecture uses support code to provide some VFP functionality. In such an implementation, only the support code can provide full details of the supported features. In this case the Media and VFP Feature registers are not used directly.

The Media and VFP Feature registers are described in:

- *Media and VFP Feature Register 0 (MVFR0)*
- *Media and VFP Feature Register 1 (MVFR1)* on page B5-38.

Media and VFP Feature Register 0 (MVFR0)

The format of the MVFR0 register is:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
VFP rounding modes	Short vectors	Square root	Divide	VFP exception trapping	Double-precision	Single-precision	A_SIMD registers

VFP rounding modes, bits [31:28]

Indicates the rounding modes supported by the VFP floating-point hardware. Permitted values are:

0b0000 Only Round to Nearest mode supported, except that Round towards Zero mode is supported for VCVT instructions that always use that rounding mode regardless of the FPSCR setting.

0b0001 All rounding modes supported.

Short vectors, bits [27:24]

Indicates the hardware support for VFP short vectors. Permitted values are:

0b0000 Not supported.

0b0001 Short vector operation supported.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

リビジョン、ビット [3:0]

浮動小数点実装の、実装定義のリビジョン番号

B5.3.2 メディアおよび VFP 機能レジスタ

メディアおよび VFP 機能レジスタは、実装にアドバンスト SIMD 拡張機能と VFP 拡張機能のいずれか、または両方が含まれている場合に、これらの拡張機能によって提供される機能を示します。これらの拡張機能の実装オプションの詳細については、*アドバンスト SIMD および VFP 拡張機能* P. A2- 20 を参照して下さい。

VFPv2 では、メディアおよび VFP 機能レジスタが実装されるかどうかは、実装定義です。

注

多くの場合、VFP アーキテクチャの完全な実装では、一部の VFP 機能を提供するためのサポートコードが使用されます。このような実装では、サポートコードのみが、サポートされている機能のすべての詳細を提供できます。この場合、メディアおよび VFP 機能レジスタは直接使用されません。

次のセクションでは、メディアおよび VFP 機能レジスタについて説明します。

- *メディアおよび VFP 機能レジスタ 0 (MVFR0)*
- *メディアおよび VFP 機能レジスタ 1(MVFR1)* P. B5- 38

メディアおよび VFP 機能レジスタ 0 (MVFR0)

MVFR0 レジスタのフォーマットは次のとおりです。

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
VFP丸め モード	ショート ベクタ	平方根	除算	VFP例外 トラップ	倍精度	単精度	A_SIMD レジスタ

VFP 丸めモード、ビット [31:28]

VFP 浮動小数点ハードウェアでサポートされている丸めモードを示します。許される値は次のとおりです。

0b0000 近似値への丸めモードのみがサポートされています。ただし、VCVT 命令は FPSCR の設定にかかわらず、常に 0 への丸めモードを使用するため、この命令専用に 0 への丸めモードがサポートされています。

0b0001 すべての丸めモードがサポートされています。

ショートベクタ、ビット [27:24]

VFP ショートベクタのハードウェアサポートを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 ショートベクタ演算がサポートされています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Square root, bits [23:20]

Indicates the hardware support for VFP square root operations. Permitted values are:

0b0000 Not supported in hardware.

0b0001 Supported.

Note

- the FSQRTS instruction also requires the single-precision VFP attribute, bits [7:4]
- the FSQRTD instruction also requires the double-precision VFP attribute, bits [11:8].

Divide, bits [19:16]

Indicates the hardware support for VFP divide operations. Permitted values are:

0b0000 Not supported in hardware.

0b0001 Supported.

Note

- the FDIVS instruction also requires the single-precision VFP attribute, bits [7:4]
- the FDIVD instruction also requires the double-precision VFP attribute, bits [11:8].

VFP exception trapping, bits [15:12]

Indicates whether the VFP hardware implementation supports exception trapping. Permitted values are:

0b0000 Not supported. This is the value for VFPv3.

0b0001 Supported by the hardware. This is the value for VFPv3U, and for VFPv2.

When exception trapping is supported, support code is needed to handle the trapped exceptions.

Note

This value does not indicate that trapped exception handling is available. Because trapped exception handling requires support code, only the support code can provide this information.

Double-precision, bits [11:8]

Indicates the hardware support for VFP double-precision operations. Permitted values are:

0b0000 Not supported in hardware.

0b0001 Supported, VFPv2.

0b0010 Supported, VFPv3.

VFPv3 adds an instruction to load a double-precision floating-point constant, and conversions between double-precision and fixed-point values.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

平方根、ビット [23:20]

VFP 平方根演算のハードウェアサポートを示します。許される値は次のとおりです。

0b0000 ハードウェアでサポートされていません。

0b0001 サポートされています。

注

- FSQRTS 命令では、単精度 VFP 属性（ビット [7:4]）も必要です。
- FSQRTD 命令では、倍精度 VFP 属性（ビット [11:8]）も必要です。

除算、ビット [19:16]

VFP 除算のハードウェアサポートを示します。許される値は次のとおりです。

0b0000 ハードウェアでサポートされていません。

0b0001 サポートされています。

注

- FDIVS 命令では、単精度 VFP 属性（ビット [7:4]）も必要です。
- FDIVD 命令では、倍精度 VFP 属性（ビット [11:8]）も必要です。

VFP 例外トラップ、ビット [15:12]

VFP ハードウェア実装で例外トラップがサポートされているかどうかを示します。許される値は次のとおりです。

0b0000 サポートされていません。これは、VFPv3 用の値です。

0b0001 ハードウェアでサポートされています。これは、VFPv3U および VFPv2 用の値です。

例外トラップがサポートされている場合は、トラップ例外を処理するサポートコードが必要です。

注

この値は、トラップ例外の処理が利用可能なことを示しているわけではありません。トラップ例外の処理にはサポートコードが必要なため、サポートコードのみが、この情報を提供できます。

倍精度、ビット [11:8]

VFP 倍精度演算のハードウェアサポートを示します。許される値は次のとおりです。

0b0000 ハードウェアでサポートされていません。

0b0001 サポートされています (VFPv2)。

0b0010 サポートされています (VFPv3)。

VFPv3 では、倍精度浮動小数点定数をロードするための命令と、倍精度値と固定小数点値との変換を行うための命令が追加されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A value of 0b0001 or 0b0010 indicates support for all VFP double-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- `FSQRTD` is only available if the Square root field is 0b0001
- `FDIVD` is only available if the Divide field is 0b0001
- conversion between double-precision and single-precision is only available if the single-precision field is nonzero.

Single-precision, bits [7:4]

Indicates the hardware support for VFP single-precision operations. Permitted values are:

0b0000 Not supported in hardware.

0b0001 Supported, VFPv2.

0b0010 Supported, VFPv3.
VFPv3 adds an instruction to load a single-precision floating-point constant, and conversions between single-precision and fixed-point values.

A value of 0b0001 or 0b0010 indicates support for all VFP single-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- `FSQRTS` is only available if the Square root field is 0b0001
- `FDIVS` is only available if the Divide field is 0b0001
- conversion between double-precision and single-precision is only available if the double-precision field is nonzero.

A_SIMD registers, bits [3:0]

Indicates support for the Advanced SIMD register bank. Permitted values are:

0b0000 Not supported.

0b0001 Supported, 16 x 64-bit registers.

0b0010 Supported, 32 x 64-bit registers.

If this field is nonzero:

- all VFP `LDC`, `STC`, `MCR`, and `MRC` instructions are supported
- if the CPUID register shows that the `MCRR` and `MRRC` instructions are supported then the corresponding VFP instructions are supported.

Media and VFP Feature Register 1 (MVFR1)

The format of the MVFR1 register is:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
Reserved, RAZ	VFP HPFP	A_SIMD HPFP	A_SIMD SPFP	A_SIMD integer	A_SIMD load/store	D_NaN mode	FtZ mode

Bits [31:28] Reserved, RAZ.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

0b0001 または 0b0010 の値は、次に示す命令を除いて、サポートされているバージョンの VFP に含まれるすべての VFP 倍精度命令がサポートされていることを示します。次に示す命令は、このフィールドが 0 でないことに加えて、記載されている条件が満たされている場合のみサポートされます。

- `FSQRTD` は、平方根フィールドが 0b0001 の場合のみ使用可能です。
- `FDIVD` は、除算フィールドが 0b0001 の場合のみ使用可能です。
- 倍精度と単精度との変換は、単精度フィールドが 0 以外の場合にのみ使用可能です。

単精度、ビット [7:4]

VFP 単精度演算のハードウェアサポートを示します。許される値は次のとおりです。

0b0000 ハードウェアでサポートされていません。

0b0001 サポートされています (VFPv2)。

0b0010 サポートされています (VFPv3)。
VFPv3 では、単精度浮動小数点定数をロードするための命令と、単精度値と固定小数点値との変換を行うための命令が追加されています。

0b0001 または 0b0010 の値は、次に示す命令を除いて、サポートされているバージョンの VFP に含まれるすべての VFP 単精度命令がサポートされていることを示します。次に示す命令は、このフィールドが 0 でないことに加えて、記載されている条件が満たされている場合のみサポートされます。

- `FSQRTS` は、平方根フィールドが 0b0001 の場合のみ使用可能です。
- `FDIVS` は、除算フィールドが 0b0001 の場合のみ使用可能です。
- 倍精度と単精度との変換は、倍精度フィールドが 0 以外の場合のみ使用可能です。

A_SIMD レジスタ、ビット [3:0]

アドバンスト SIMD レジスタバンクのサポートを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされています（16×64 ビットレジスタ）。

0b0010 サポートされています（32×64 ビットレジスタ）。

このフィールドが 0 以外の場合、次の命令がサポートされます。

- VFP `LDC`、`STC`、`MCR`、`MRC` のすべての命令がサポートされます。
- CPUID レジスタが、`MCRR` 命令と `MRRC` 命令がサポートされていることを示している場合は、対応する VFP 命令がサポートされます。

メディアおよび VFP 機能レジスタ 1(MVFR1)

MVFR1 レジスタのフォーマットは次のとおりです。

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
予約、RAZ	VFP HPFP	A_SIMD HPFP	A_SIMD SPFP	A_SIMD 整数	A_SIMD ロード/ストア	D_NaN モード	FtZ mode

ビット [31:28] 予約、RAZ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

VFP HPFP, bits[27:24]

Indicates whether the VFP supports half-precision floating-point conversion operations. Permitted values are:

0b0000 Not supported.

0b0001 Supported.

A_SIMD HPFP, bits[23:20]

Indicates whether Advanced SIMD supports half-precision floating-point conversion operations. Permitted values are:

0b0000 Not supported.

0b0001 Supported. This value is only permitted if the A_SIMD SPFP field is 0b0001.

A_SIMD SPFP, bits [19:16]

Indicates whether the Advanced SIMD extension supports single-precision floating-point operations. Permitted values are:

0b0000 Not supported.

0b0001 Supported. This value is only permitted if the A_SIMD integer field is 0b0001.

A_SIMD integer, bits [15:12]

Indicates whether the Advanced SIMD extension supports integer operations. Permitted values are:

0b0000 Not supported.

0b0001 Supported.

A_SIMD load/store, bits [11:8]

Indicates whether the Advanced SIMD extension supports load/store instructions. Permitted values are:

0b0000 Not supported.

0b0001 Supported.

D_NaN mode, bits [7:4]

Indicates whether the VFP hardware implementation supports only the Default NaN mode. Permitted values are:

0b0000 Hardware supports only the Default NaN mode. If a VFP subarchitecture is implemented its support code might include support for propagation of NaN values.

0b0001 Hardware supports propagation of NaN values.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

VFP HPFP、ビット [27:24]

VFP で半精度浮動小数点変換演算がサポートされているかどうかを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされています。

A_SIMD HPFP、ビット [23:20]

アドバンスト SIMD で半精度浮動小数点変換演算がサポートされているかどうかを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされています。この値は、A_SIMD SPFP フィールドが 0b0001 の場合にのみ許されます。

A_SIMD SPFP、ビット [19:16]

アドバンスト SIMD 拡張機能で、単精度浮動小数点演算がサポートされているかどうかを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされています。この値は、A_SIMD integer フィールドが 0b0001 の場合にのみ許されます。

A_SIMD 整数、ビット [15:12]

アドバンスト SIMD 拡張機能で整数演算がサポートされているかどうかを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされています。

A_SIMD ロード / ストア、ビット [11:8]

アドバンスト SIMD 拡張機能でロード / ストア命令がサポートされているかどうかを示します。許される値は次のとおりです。

0b0000 サポートされていません。

0b0001 サポートされています。

D_NaN モード、ビット [7:4]

VFP ハードウェア実装で、デフォルト NaN モードのみがサポートされているかどうかを示します。許される値は次のとおりです。

0b0000 ハードウェアでデフォルト NaN モードのみがサポートされています。VFP サブアーキテクチャが実装されている場合は、そのサポートコードに NaN 値の伝播のサポートが含まれている可能性があります。

0b0001 ハードウェアで NaN 値の伝播がサポートされています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

FtZ mode, bits [7:4]

Indicates whether the VFP hardware implementation supports only the Flush-to-Zero mode of operation. Permitted values are:

0b0000 Hardware supports only the Flush-to-Zero mode of operation. If a VFP subarchitecture is implemented its support code might include support for full denormalized number arithmetic.

0b0001 Hardware supports full denormalized number arithmetic.

B5.3.3 Accessing the Advanced SIMD and VFP feature identification registers

You access the Advanced SIMD and VFP feature identification registers using the `VMRS` instruction, see *VMRS* on page A8-658.

For example:

```
VMRS <Rt>, FPSID    ; Read Floating-Point System ID Register
VMRS <Rt>, MVFR1    ; Read Media and VFP Feature Register 1
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

FtZ モード、ビット [7:4]

VFP ハードウェア実装で、演算の Flush-to-Zero モードのみがサポートされているかどうかを示します。許される値は次のとおりです。

0b0000 ハードウェアで、演算の Flush-to-Zero モードのみがサポートされています。VFP サブアーキテクチャが実装されている場合は、そのサポートコードに完全な非正規化数演算のサポートが含まれている可能性があります。

0b0001 ハードウェアで完全な非正規化数演算がサポートされています。

B5.3.3 アドバンスト SIMD および VFP 機能識別レジスタへのアクセス

アドバンスト SIMD および VFP 機能識別レジスタには、`VMRS` 命令を使用してアクセスします。*VMRS* P. A8-658 を参照して下さい。

次に例を示します。

```
VMRS <Rt>, FPSID    ; Read Floating-Point System ID Register
VMRS <Rt>, MVFR1    ; Read Media and VFP Feature Register 1
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter B6
System Instructions

This chapter describes the instructions that are only available, or that behave differently, in privileged modes. It contains the following section:

- *Alphabetical list of instructions* on page B6-2.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 B6 章
システム命令

本章では、特権モードでのみ使用できる、または特権モードでは動作が異なる命令について説明します。本章は次のセクションから構成されています。

- *命令のアルファベット順リスト* P. B6-2

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1 Alphabetical list of instructions

This section lists every instruction that behaves differently in privileged modes, or that is only available in privileged modes. For information about privileged modes see *ARM processor modes and core registers* on page B1-6.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B6.1 命令のアルファベット順リスト

このセクションでは、特権モードでは動作が異なる、または特権モードでのみ使用できる、すべての命令を列挙します。特権モードの詳細については、*ARM プロセッサモードとコアレジスタ* P. B1-6 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.1 CPS

Change Processor State is available only in privileged modes. It changes one or more of the A, I, and F interrupt disable bits and the mode bits of the CPSR, without changing the other CPSR bits.

Encoding T1 ARMv6*, ARMv7

```
CPS<effect> <iflags>                                   Not permitted in IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	A	I	F

```
enable = (im == '0');  disable = (im == '1');  changemode = FALSE;
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if InITBlock() then UNPREDICTABLE;
```

Encoding T2 ARMv6T2, ARMv7

```
CPS<effect>.W <iflags>{,#<mode>}                       Not permitted in IT block.
CPS #<mode>                                            Not permitted in IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	imod		M	A	I	F	mode				

```
if imod == '00' && M == '0' then SEE "Hint instructions";
enable = (imod == '10');  disable = (imod == '11');  changemode = (M == '1');
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if imod == '01' || InITBlock() then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
CPS<effect> <iflags>{,#<mode>}
CPS #<mode>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	0	imod		M	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	A	I	F	0	mode				

```
enable = (imod == '10');  disable = (imod == '11');  changemode = (M == '1');
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if (imod == '00' && M == '0') || imod == '01' then UNPREDICTABLE;
```

Assembler syntax

```
CPS<effect><q>  <iflags> {, #<mode>}
CPS<q>  #<mode>
```

where:

`<effect>` The effect required on the A, I, and F bits in the CPSR. This is one of:

- `IE` Interrupt Enable. This sets the specified bits to 0.
- `ID` Interrupt Disable. This sets the specified bits to 1.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.1 CPS

CPS（プロセッサ状態変更）は、特権モードでのみ使用できます。この命令は、A、I、F の割り込み禁止ビットの 1 つ以上を変更し、また、CPSR の他のビットを変更せずモードビットのみを変更します。

エンコード T1 ARMv6*、ARMv7

```
CPS<effect> <iflags>                                   IT ブロック内部では許可されない
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	A	I	F

```
enable = (im == '0');  disable = (im == '1');  changemode = FALSE;
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if InITBlock() then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

```
CPS<effect>.W <iflags>{,#<mode>}                       IT ブロック内部では許可されない
CPS #<mode>                                            IT ブロック内部では許可されない
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	imod		M	A	I	F	mode				

```
if imod == '00' && M == '0' then SEE "Hint instructions";
enable = (imod == '10');  disable = (imod == '11');  changemode = (M == '1');
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if imod == '01' || InITBlock() then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
CPS<effect> <iflags>{,#<mode>}
CPS #<mode>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	0	imod		M	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	A	I	F	0	mode				

```
enable = (imod == '10');  disable = (imod == '11');  changemode = (M == '1');
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if (imod == '00' && M == '0') || imod == '01' then UNPREDICTABLE;
```

アセンブラ構文

```
CPS<effect><q>  <iflags> {, #<mode>}
CPS<q>  #<mode>
```

各項目の説明については以下を参照して下さい。

`<effect>` CPSR の A、I、F ビットに対する作用。次のいずれかを指定します。

- `IE` 割り込み許可。指定のビットが 0 にセットされます。
- `ID` 割り込み禁止。指定のビットが 1 にセットされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If <effect> is specified, the bits to be affected are specified by <iflags>. The mode can optionally be changed by specifying a mode number as <mode>.

If <effect> is not specified, then:

- <iflags> is not specified and interrupt settings are not changed
- <mode> specifies the new mode number.

<q> See *Standard assembler syntax fields* on page A8-7. A CPS instruction must be unconditional.

<iflags> Is a sequence of one or more of the following, specifying which interrupt disable flags are affected:

- a Sets the A bit in the instruction, causing the specified effect on the CPSR.A (asynchronous abort) bit.
- i Sets the I bit in the instruction, causing the specified effect on the CPSR.I (IRQ interrupt) bit.
- f Sets the F bit in the instruction, causing the specified effect on the CPSR.F (FIQ interrupt) bit.

<mode> The number of the mode to change to. If this option is omitted, no mode change occurs.

Operation

```
EncodingSpecificOperations();
if CurrentModeIsPrivileged() then
    cpsr_val = CPSR;
    if enable then
        if affectA then cpsr_val<8> = '0';
        if affectI then cpsr_val<7> = '0';
        if affectF then cpsr_val<6> = '0';
    if disable then
        if affectA then cpsr_val<8> = '1';
        if affectI then cpsr_val<7> = '1';
        if affectF then cpsr_val<6> = '1';
    if changemode then
        cpsr_val<4:0> = mode;
    CPSRWriteByInstr(cpsr_val, '1111', TRUE);
```

Exceptions

None.

Hint instructions

If the imod field and the M bit in encoding T2 are '00' and'0' respectively, a hint instruction is encoded. To determine which hint instruction, see *Change Processor State, and hints* on page A6-21.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

<effect> が指定された場合は、影響を受けるビットが <iflags> によって指定されます。オプションとして、<mode> としてモード番号を指定することによって、モードを変更できます。

<effect> が指定されない場合は、次のようになります。

- <iflags> は指定されず、割り込み設定は変更されません。
- <mode> で新しいモード番号が指定されます。

<q> 詳細については、*標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。CPS 命令は必ず無条件です。

<iflags> 影響を受ける割り込みディセーブルフラグを指定する、次の 1 つ以上のシーケンスです。

- a 命令の A ビットをセットします。CPSR.A（非同期アボート）ビットに対して、指定された作用が実行されます。
- i 命令の I ビットをセットします。CPSR.I（IRQ 割り込み）ビットに対して、指定された作用が実行されます。
- f 命令の F ビットをセットします。CPSR.F（FIQ 割り込み）ビットに対して、指定された作用が実行されます。

<mode> 変更するモード番号。このオプションを省略した場合は、モード変更が行われません。

動作

```
EncodingSpecificOperations();
if CurrentModeIsPrivileged() then
    cpsr_val = CPSR;
    if enable then
        if affectA then cpsr_val<8> = '0';
        if affectI then cpsr_val<7> = '0';
        if affectF then cpsr_val<6> = '0';
    if disable then
        if affectA then cpsr_val<8> = '1';
        if affectI then cpsr_val<7> = '1';
        if affectF then cpsr_val<6> = '1';
    if changemode then
        cpsr_val<4:0> = mode;
    CPSRWriteByInstr(cpsr_val, '1111', TRUE);
```

例外

なし

ヒント命令

エンコード T2 で、imod フィールドと M ビットがそれぞれ '00' と '0' の場合は、ヒント命令がエンコードされます。どのヒント命令かを判定するには、*プロセッサ状態の変更とヒント* P. A6-21 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.2 LDM (exception return)

Load Multiple (exception return) loads multiple registers from consecutive memory locations using an address from a base register. The SPSR of the current mode is copied to the CPSR. An address adjusted by the size of the data loaded can optionally be written back to the base register.

The registers loaded include the PC. The word loaded for the PC is treated as an address and a branch occurs to that address.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDM{<amode>}<c> <Rn>{!},<registers_with_pc>^
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0	P	U	1	W	1	Rn	1	register_list

```
n = UInt(Rn);  registers = register_list;
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
if wback && registers<n> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```

Assembler syntax

```
LDM{<amode>}<c><q>  <Rn>{!}, <registers_with_pc>^
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<amode>` is one of:

- `DA` Decrement After. The consecutive memory addresses end at the address in the base register. For this instruction, `FA`, meaning Full Ascending, is equivalent to `DA`. Encoded as P = 0, U = 0.
- `DB` Decrement Before. The consecutive memory addresses end one word below the address in the base register. For this instruction, `EA`, meaning Empty Ascending, is equivalent to `DB`. Encoded as P = 1, U = 0.
- `IA` Increment After. The consecutive memory addresses start at the address in the base register. This is the default, and is normally omitted. For this instruction, `FD`, meaning Full Descending, is equivalent to `IA`. Encoded as P = 0, U = 1.
- `IB` Increment Before. The consecutive memory addresses start one word above the address in the base register. For this instruction, `ED`, meaning Empty Descending, is equivalent to `IB`. Encoded as P = 1, U = 1.

`<Rn>` The base register. This register can be the SP.

`!` Causes the instruction to write a modified value back to `<Rn>`. Encoded as W = 1.

If `!` is omitted, the instruction does not change `<Rn>` in this way. Encoded as W = 0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B6.1.2 LDM（例外からの復帰）

LDM（複数ロード）（例外からの復帰）は、ベースレジスタからのアドレスを使用して、連続するメモリ位置から複数のレジスタをロードします。現在のモードの SPSR が CPSR にコピーされます。オプションとして、ロードされるデータサイズによって調整されたアドレスを、ベースレジスタに書き戻すことができます。

ロードされるレジスタには PC が含まれます。PC に対してロードされたワードはアドレスとして扱われ、そのアドレスへの分岐が実行されます。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDM{<amode>}<c> <Rn>{!},<registers_with_pc>^
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0	P	U	1	W	1	Rn	1	register_list

```
n = UInt(Rn);  registers = register_list;
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
if wback && registers<n> == '1' && ArchVersion() >= 7 then UNPREDICTABLE;
```

アセンブラ構文

```
LDM{<amode>}<c><q>  <Rn>{!}, <registers_with_pc>^
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<amode>` 次のいずれかを指定します。

- `DA` ポストデクリメント。連続するメモリアドレスは、ベースレジスタのアドレスで終わります。この命令の場合は、フル上昇を意味する `FA` と、`DA` が等価です。P = 0、U = 0 としてエンコードされます。
- `DB` プリデクリメント。連続するメモリアドレスは、ベースレジスタのアドレスの 1 ワード前で終わります。この命令の場合は、空上昇を意味する `EA` と、`DB` が等価です。P = 1、U = 0 としてエンコードされます。
- `IA` ポストインクリメント。連続するメモリアドレスは、ベースレジスタのアドレスで始まります。これはデフォルトで、通常は省略されます。この命令の場合は、フル下降を意味する `FD` と、`IA` が等価です。P = 0、U = 1 としてエンコードされます。
- `IB` プリインクリメント。連続するメモリアドレスは、ベースレジスタのアドレスの 1 ワード後から始まります。この命令の場合は、空下降を意味する `ED` と、`IB` が等価です。P = 1、U = 1 としてエンコードされます。

`<Rn>` ベースレジスタ。SP を指定できます。

`!` 命令の実行によって変更された値を `<Rn>` に書き戻すように指示します。W = 1 としてエンコードされます。

`!` を省略した場合、命令によって `<Rn>` は変更されません。W = 0 としてエンコードされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<registers_with_pc>

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must be specified in the register list, and the instruction causes a branch to the address (data) loaded into the PC.

The pre-UAL syntax LDM<c>{<amode>} is equivalent to LDM{<amode>}<c>.

Note

Instructions with similar syntax but without the PC included in <registers> are described in *LDM (user registers)* on page B6-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentModeIsUserOrSystem() then UNPREDICTABLE;
    length = 4*BitCount(registers) + 4;
    address = if increment then R[n] else R[n]-length;
    if wordhigher then address = address+4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    new_pc_value = MemA[address,4];
    if wback && registers<n> == '0' then R[n] = if increment then R[n]+length else R[n]-length;
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
    CPSRWriteByInstr(SPSR[], '1111', TRUE);
    BranchWritePC(new_pc_value);
```

Exceptions

Data Abort.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

<registers_with_pc>

1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストです。ロードするレジスタのセットを指定します。最下位のメモリアドレスに置かれているデータは最も小さい番号のレジスタにロードされ、以下順に、最上位のメモリアドレスのデータは最も大きい番号のレジスタにロードされます。レジスタリストには PC を指定する必要があり、命令によって、PC にロードされたアドレス（データ）への分岐が実行されます。

UAL 以前の構文の LDM<c>{<amode>} は、LDM{<amode>}<c> と等価です。

注

同様の構文で、<registers> に PC が含まれない命令については、*LDM（ユーザレジスタ）* P. B6-7 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentModeIsUserOrSystem() then UNPREDICTABLE;
    length = 4*BitCount(registers) + 4;
    address = if increment then R[n] else R[n]-length;
    if wordhigher then address = address+4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    new_pc_value = MemA[address,4];
    if wback && registers<n> == '0' then R[n] = if increment then R[n]+length else R[n]-length;
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
    CPSRWriteByInstr(SPSR[], '1111', TRUE);
    BranchWritePC(new_pc_value);
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.3 LDM (user registers)

Load Multiple (user registers) is UNPREDICTABLE in User or System modes. In exception modes, it loads multiple User mode registers from consecutive memory locations using an address from a banked base register. Writeback to the base register is not available with this instruction.

The registers loaded cannot include the PC.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDM{<amode>}<c> <Rn>,<registers_without_pc>^
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0	P	U	1	(0)	1	Rn	0	register_list

```
n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

Assembler syntax

```
LDM{<amode>}<c><q>  <Rn>, <registers_without_pc>^
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<amode>` is one of:

- `DA` Decrement After. The consecutive memory addresses end at the address in the base register. For this instruction, `FA`, meaning Full Ascending, is equivalent to `DA`. Encoded as P = 0, U = 0.
- `DB` Decrement Before. The consecutive memory addresses end one word below the address in the base register. For this instruction, `EA`, meaning Empty Ascending, is equivalent to `DB`. Encoded as P = 1, U = 0.
- `IA` Increment After. The consecutive memory addresses start at the address in the base register. This is the default, and is normally omitted. For this instruction, `FD`, meaning Full Descending, is equivalent to `IA`. Encoded as P = 0, U = 1.
- `IB` Increment Before. The consecutive memory addresses start one word above the address in the base register. For this instruction, `ED`, meaning Empty Descending, is equivalent to `IB`. Encoded as P = 1, U = 1.

`<Rn>` The base register. This register can be the SP.

`<registers_without_pc>`

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the `LDM` instruction. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must not be in the register list.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.3 LDM（ユーザレジスタ）

LDM（複数ロード）（ユーザレジスタ）は、ユーザモードまたはシステムモードでは予測不能です。例外モードでは、バンクされたベースレジスタからのアドレスを使用して、連続するメモリ位置から複数のユーザモード レジスタをロードします。この命令では、ベースレジスタへのライトバックを使用できません。

ロードされるレジスタに PC を含めることはできません。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
LDM{<amode>}<c> <Rn>,<registers_without_pc>^
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0	P	U	1	(0)	1	Rn	0	register_list

```
n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

アセンブラ構文

```
LDM{<amode>}<c><q>  <Rn>, <registers_without_pc>^
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<amode>` 次のいずれかを指定します。

- `DA` ポストデクリメント。連続するメモリアドレスは、ベースレジスタのアドレスで終わります。この命令の場合は、フル上昇を意味する `FA` と、`DA` が等価です。P = 0、U = 0 としてエンコードされます。
- `DB` プリデクリメント。連続するメモリアドレスは、ベースレジスタのアドレスの 1 ワード前で終わります。この命令の場合は、空上昇を意味する `EA` と、`DB` が等価です。P = 1、U = 0 としてエンコードされます。
- `IA` ポストインクリメント。連続するメモリアドレスは、ベースレジスタのアドレスで始まります。これはデフォルトで、通常は省略されます。この命令の場合は、フル下降を意味する `FD` と、`IA` が等価です。P = 0、U = 1 としてエンコードされます。
- `IB` プリインクリメント。連続するメモリアドレスは、ベースレジスタのアドレスの 1 ワード後から始まります。この命令の場合は、空下降を意味する `ED` と、`IB` が等価です。P = 1、U = 1 としてエンコードされます。

`<Rn>` ベースレジスタ。SP を指定できます。

`<registers_without_pc>`

1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストです。`LDM` 命令でロードするレジスタのセットを指定します。最下位のメモリアドレスに置かれているデータは最も小さい番号のレジスタにロードされ、以下順に、最上位のメモリアドレスのデータは最も大きい番号のレジスタにロードされます。PC をレジスタリストに含めることはできません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The pre-UAL syntax LDM<c>{<amode>} is equivalent to LDM{<amode>}<c>.

Note

Instructions with similar syntax but with the PC included in <registers> are described in *LDM (exception return)* on page B6-5.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentModeIsUserOrSystem() then UNPREDICTABLE;
    length = 4*BitCount(registers);
    address = if increment then R[n] else R[n]-length;
    if wordhigher then address = address+4;
    for i = 0 to 14
        if registers<i> == '1' then  // Load User mode ('10000') register
            Rmode[i, '10000'] = MemA[address,4];  address = address + 4;
```

Exceptions

Data Abort.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

UAL 以前の構文の LDM<c>{<amode>} は、LDM{<amode>}<c> と等価です。

注

同様の構文で、<registers> に PC が含まれる命令については、*LDM（例外からの復帰）* P. B6-5 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentModeIsUserOrSystem() then UNPREDICTABLE;
    length = 4*BitCount(registers);
    address = if increment then R[n] else R[n]-length;
    if wordhigher then address = address+4;
    for i = 0 to 14
        if registers<i> == '1' then  // Load User mode ('10000') register
            Rmode[i, '10000'] = MemA[address,4];  address = address + 4;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.4 LDRBT, LDRHT, LDRSBT, LDRSHT, and LDRT

Even in privileged modes, loads from memory by these instructions are restricted in the same way as loads from memory in User mode. This is encapsulated in the `MemA_unpriv[]` and `MemU_unpriv[]` pseudocode functions. For details see *Aligned memory accesses* on page B2-31 and *Unaligned memory accesses* on page B2-32.

For details of the instructions see:

- *LDRBT* on page A8-134
- *LDRHT* on page A8-158
- *LDRSBT* on page A8-166
- *LDRSHT* on page A8-174
- *LDRT* on page A8-176.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.4 LDRBT、LDRHT、LDRSBT、LDRSHT、LDRT

これらの命令によるメモリからのロードは、特権モードでも、ユーザモードでのメモリからのロードと同様に制限されます。これは、`MemA_unpriv[]` および `MemU_unpriv[]` 擬似コード関数にカプセル化されます。詳細については、*アラインド メモリアクセス* P. B2-31 と *アンアラインドメモリアクセス* P. B2-32 を参照して下さい。

命令の詳細については、次の項目を参照して下さい。

- *LDRBT* P. A8-134
- *LDRHT* P. A8-158
- *LDRSBT* P. A8-166
- *LDRSHT* P. A8-174
- *LDRT* P. A8-176

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.5 MRS

Move to Register from Special Register moves the value from the CPSR or SPSR of the current mode into a general-purpose register.

Encoding T1 ARMv6T2, ARMv7

```
MRS<c> <Rd>,<spec_reg>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	1	R	(1)	(1)	(1)	(1)	1	0	(0)	0	Rd				(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
d = UInt(Rd);  read_spsr = (R == '1');
if BadReg(d) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MRS<c> <Rd>,<spec_reg>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	R	0	0	(1)	(1)	(1)	(1)	Rd				(0)	(0)	(0)	(0)	0	0	0	0	(0)	(0)	(0)	(0)

```
d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.5 MRS

MRS（特殊レジスタからレジスタへの移動）は、現在のモードの CPSR または SPSR の値を、汎用レジスタに移動します。

エンコード T1 ARMv6T2、ARMv7

```
MRS<c> <Rd>,<spec_reg>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	1	R	(1)	(1)	(1)	(1)	1	0	(0)	0	Rd				(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
d = UInt(Rd);  read_spsr = (R == '1');
if BadReg(d) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MRS<c> <Rd>,<spec_reg>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	R	0	0	(1)	(1)	(1)	(1)	Rd				(0)	(0)	(0)	(0)	0	0	0	0	(0)	(0)	(0)	(0)

```
d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
MRS<c><q>  <Rd>, <spec_reg>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rd>` The destination register.

`<spec_reg>` Is one of:

- `APSR`
- `CPSR`
- `SPSR`.

ARM recommends the `APSR` form when only the N, Z, C, V, Q, or GE[3:0] bits of the read value are going to be used (see *The Application Program Status Register (APSR)* on page A2-14).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if read_spsr then
        if CurrentModeIsUserOrSystem() then
            UNPREDICTABLE;
        else
            R[d] = SPSR[];
    else
        // CPSR is read with execution state bits other than E masked out.
        R[d] = CPSR AND '11111000 11111111 00000011 11011111';
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
MRS<c><q>  <Rd>, <spec_reg>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rd>` デスティネーションレジスタ

`<spec_reg>` 次のいずれかを指定します。

- `APSR`
- `CPSR`
- `SPSR`

読み出し値の N、Z、C、V、Q、または GE[3:0] ビットのみが使用される場合は、`APSR` 形式をお勧めします。*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if read_spsr then
        if CurrentModeIsUserOrSystem() then
            UNPREDICTABLE;
        else
            R[d] = SPSR[];
    else
        // CPSR is read with execution state bits other than E masked out.
        R[d] = CPSR AND '11111000 11111111 00000011 11011111';
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.6 MSR (immediate)

Move immediate value to Special Register moves selected bits of an immediate value to the CPSR or the SPSR of the current mode.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MSR<c> <spec_reg>,#<const>
```

31 30 29 28	27 26 25 24 23	22	21 20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0 1 1 0	R	1 0	mask	(1) (1) (1) (1)	imm12

```
if mask == '0000' && R == '0' then SEE "Related encodings";
imm32 = ARMExpandImm(imm12);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

Assembler syntax

```
MSR<c><q>  <spec_reg>, #<const>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<spec_reg>` Is one of:

- `APSR_<bits>`
- `CPSR_<fields>`
- `SPSR_<fields>`.

ARM recommends the `APSR` forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see *The Application Program Status Register (APSR)* on page A2-14.

`<const>` The immediate value to be transferred to `<spec_reg>`. See *Modified immediate constants in ARM instructions* on page A5-9 for the range of values.

`<bits>` Is one of `nzcvq`, `g`, or `nzcvqg`.

In the A and R profiles:

- `APSR_nzcvq` is the same as `CPSR_f` (mask == '1000')
- `APSR_g` is the same as `CPSR_s` (mask == '0100')
- `APSR_nzcvqg` is the same as `CPSR_fs` (mask == '1100').

`<fields>` Is a sequence of one or more of the following:

c	mask<0> = '1' to enable writing of bits<7:0> of the destination PSR
x	mask<1> = '1' to enable writing of bits<15:8> of the destination PSR
s	mask<2> = '1' to enable writing of bits<23:16> of the destination PSR
f	mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B6.1.6 MSR（イミディエート）

MSR（イミディエート値の特殊レジスタへの移動）は、イミディエート値の選択されたビットを、現在のモードの CPSR または SPSR に移動します。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MSR<c> <spec_reg>,#<const>
```

31 30 29 28	27 26 25 24 23	22	21 20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0 1 1 0	R	1 0	mask	(1) (1) (1) (1)	imm12

```
if mask == '0000' && R == '0' then SEE "Related encodings";
imm32 = ARMExpandImm(imm12);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

アセンブラ構文

```
MSR<c><q>  <spec_reg>, #<const>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<spec_reg>` 次のいずれかを指定します。

- `APSR_<bits>`
- `CPSR_<fields>`
- `SPSR_<fields>`

N、Z、C、V、Q、GE[3:0] ビットのみを書き込む場合は、`APSR` 形式をお勧めします。詳細については、*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。

`<const>` `<spec_reg>` に転送されるイミディエート値。値の範囲については、*ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

`<bits>` `nzcvq`、`g`、`nzcvqg` のいずれかを指定します。

A プロファイルと R プロファイルでは、次の条件が適用されます。

- `APSR_nzcvq` は `CPSR_f` (mask == '1000') と同じです。
- `APSR_g` は `CPSR_s` (mask == '0100') と同じです。
- `APSR_nzcvqg` は `CPSR_fs` (mask == '1100') と同じです。

`<fields>` 次のうち 1 つまたは複数のシーケンス。

c	mask〈0〉 = '1' で、デスティネーション PSR のビット 〈7:0〉 の書き込みを可能にします。
x	mask〈1〉 = '1' で、デスティネーション PSR のビット 〈15:8〉 の書き込みを可能にします。
s	mask〈2〉 = '1' で、デスティネーション PSR のビット 〈23:16〉 の書き込みを可能にします。
f	mask〈3〉 = '1' で、デスティネーション PSR のビット 〈31:24〉 の書き込みを可能にします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        SPSRWriteByInstr(imm32, mask);
    else
        CPSRWriteByInstr(imm32, mask, FALSE);  // Does not affect execution state bits
                                               // other than E
```

Exceptions

None.

E bit

The CPSR.E bit is writable from any mode using an MSR instruction. Use of this to change its value is deprecated. Use the SETEND instruction instead.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        SPSRWriteByInstr(imm32, mask);
    else
        CPSRWriteByInstr(imm32, mask, FALSE);  // Does not affect execution state bits
                                               // other than E
```

例外

なし

E ビット

CPSR.E ビットは、任意のモードから MSR 命令を使用して書き込むことができます。この命令を使用した値の変更は非推奨です。代わりに、SETEND 命令を使用して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.7 MSR (register)

Move to Special Register from ARM core register moves the value of a general-purpose register to the CPSR or the SPSR of the current mode.

Encoding T1 ARMv6T2, ARMv7

```
MSR<c> <spec_reg>,<Rn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	R	Rn				1	0	(0)	0	mask				(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
n = UInt(Rn);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if BadReg(n) then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MSR<c> <spec_reg>,<Rn>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	R	1	0	mask				(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	Rn			

```
n = UInt(Rn);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

Assembler syntax

```
MSR<c><q>  <spec_reg>, <Rn>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<spec_reg>` Is one of:
- `APSR_<bits>`
- `CPSR_<fields>`
- `SPSR_<fields>`.

ARM recommends the `APSR` forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see *The Application Program Status Register (APSR)* on page A2-14.

`<Rn>` Is the general-purpose register to be transferred to `<spec_reg>`.

`<bits>` Is one of `nzcvq`, `g`, or `nzcvqg`.

In the A and R profiles:
- `APSR_nzcvq` is the same as `CPSR_f` (mask == '1000')
- `APSR_g` is the same as `CPSR_s` (mask == '0100')
- `APSR_nzcvqg` is the same as `CPSR_fs` (mask == '1100').

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B6.1.7 MSR（レジスタ）

MSR（ARM コアレジスタから特殊レジスタへの移動）は、汎用レジスタの値を、現在のモードの CPSR または SPSR へ移動します。

エンコード T1 ARMv6T2、ARMv7

```
MSR<c> <spec_reg>,<Rn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	R	Rn				1	0	(0)	0	mask				(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
n = UInt(Rn);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if BadReg(n) then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
MSR<c> <spec_reg>,<Rn>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	R	1	0	mask				(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	Rn			

```
n = UInt(Rn);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

アセンブラ構文

```
MSR<c><q>  <spec_reg>, <Rn>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<spec_reg>` 次のいずれかを指定します。
- `APSR_<bits>`
- `CPSR_<fields>`
- `SPSR_<fields>`

N、Z、C、V、Q、GE[3:0] ビットのみを書き込む場合は、`APSR` 形式をお勧めします。詳細については、*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。

`<Rn>` `<spec_reg>` に転送される汎用レジスタ

`<bits>` `nzcvq`、`g`、`nzcvqg` のいずれかを指定します。

A プロファイルと R プロファイルでは、次の条件が適用されます。
- `APSR_nzcvq` は `CPSR_f` (mask == '1000') と同じです。
- `APSR_g` は `CPSR_s` (mask == '0100') と同じです。
- `APSR_nzcvqg` は `CPSR_fs` (mask == '1100') と同じです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<fields> Is a sequence of one or more of the following:

- c mask<0> = '1' to enable writing of bits<7:0> of the destination PSR
- x mask<1> = '1' to enable writing of bits<15:8> of the destination PSR
- s mask<2> = '1' to enable writing of bits<23:16> of the destination PSR
- f mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        SPSRWriteByInstr(R[n], mask);
    else
        CPSRWriteByInstr(R[n], mask, FALSE);  // Does not affect execution state bits
                                              // other than E
```

Exceptions

None.

E bit

The CPSR.E bit is writable from any mode using an MSR instruction. Use of this to change its value is deprecated. Use the SETEND instruction instead.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<fields> 次のうち 1 つまたは複数のシーケンス。

- c mask〈0〉= '1' で、デスティネーション PSR のビット〈7:0〉の書き込みを可能にします。
- x mask〈1〉= '1' で、デスティネーション PSR のビット〈15:8〉の書き込みを可能にします。
- s mask〈2〉= '1' で、デスティネーション PSR のビット〈23:16〉の書き込みを可能にします。
- f mask〈3〉= '1' で、デスティネーション PSR のビット〈31:24〉の書き込みを可能にします。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        SPSRWriteByInstr(R[n], mask);
    else
        CPSRWriteByInstr(R[n], mask, FALSE);  // Does not affect execution state bits
                                              // other than E
```

例外

なし

E ビット

CPSR.E ビットは、任意のモードから MSR 命令を使用して書き込むことができます。この命令を使用した値の変更は非推奨です。代わりに、SETEND 命令を使用して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.8 RFE

Return From Exception loads the PC and the CPSR from the word at the specified address and the following word respectively. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
RFEDB<c> <Rn>{!}
```

Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	0	W	1	Rn				(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
n = UInt(Rn);  wback = (W == '1');  increment = FALSE;  wordhigher = FALSE;
if n == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T2 ARMv6T2, ARMv7

```
RFE{IA}<c> <Rn>{!}
```

Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	1	0	W	1	Rn				(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
n = UInt(Rn);  wback = (W == '1');  increment = TRUE;  wordhigher = FALSE;
if n == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
RFE{<amode>} <Rn>{!}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	P	U	0	W	1	Rn				(0)	(0)	(0)	(0)	(1)	(0)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
n = UInt(Rn);
wback = (W == '1');  inc = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.8 RFE

RFE（例外からの復帰）は、指定されたアドレスに存在するワードとその次のワードから、それぞれ PC と CPSR をロードします。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
RFEDB<c> <Rn>{!}
```

IT ブロック外部または IT ブロック内部の最後

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	0	W	1	Rn				(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
n = UInt(Rn);  wback = (W == '1');  increment = FALSE;  wordhigher = FALSE;
if n == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

```
RFE{IA}<c> <Rn>{!}
```

IT ブロック外部または IT ブロック内部の最後

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	1	0	W	1	Rn				(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
n = UInt(Rn);  wback = (W == '1');  increment = TRUE;  wordhigher = FALSE;
if n == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
RFE{<amode>} <Rn>{!}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	P	U	0	W	1	Rn				(0)	(0)	(0)	(0)	(1)	(0)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
n = UInt(Rn);
wback = (W == '1');  inc = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
RFE{<amode>}<c><q>  <Rn>{!}
```

where:

`<amode>` is one of:

- `DA` Decrement After. ARM code only. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0 in encoding A1.
- `DB` Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoding T1, or encoding A1 with P = 1, U = 0.
- `IA` Increment After. The consecutive memory addresses start at the address in the base register. This is the default, and is normally omitted. Encoding T2, or encoding A1 with P = 0, U = 1.
- `IB` Increment Before. ARM code only. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1 in encoding A1.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `RFE` instruction must be unconditional.

`<Rn>` The base register.

`!` Causes the instruction to write a modified value back to `<Rn>`. If `!` is omitted, the instruction does not change `<Rn>`.

`RFEFA`, `RFEEA`, `RFEFD`, and `RFEED` are pseudo-instructions for `RFEDA`, `RFEDB`, `RFEIA`, and `RFEIB` respectively, referring to their use for popping data from Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !CurrentModeIsPrivileged() || CurrentInstrSet() == InstrSet_ThumbEE then
        UNPREDICTABLE;
    else
        address = if increment then R[n] else R[n]-8;
        if wordhigher then address = address+4;
        CPSRWriteByInstr(MemA[address+4,4], '1111', TRUE);
        BranchWritePC(MemA[address,4]);
        if wback then R[n] = if increment then R[n]+8 else R[n]-8;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
RFE{<amode>}<c><q>  <Rn>{!}
```

各項目の説明については以下を参照して下さい。

`<amode>` 次のいずれかを指定します。

- `DA` ポストデクリメント。ARM コードのみ。連続するメモリアドレスは、ベースレジスタのアドレスで終わります。エンコード A1 の P = 0、U = 1 としてエンコードされます。
- `DB` プリデクリメント。連続するメモリアドレスは、ベースレジスタのアドレスの 1 ワード前で終わります。エンコード T1、または P = 1、U = 0 のエンコード A1。
- `IA` ポストインクリメント。連続するメモリアドレスは、ベースレジスタのアドレスで始まります。これはデフォルトで、通常は省略されます。エンコード T2、または P = 0、U = 1 のエンコード A1。
- `IB` プリインクリメント。ARM コードのみ。連続するメモリアドレスは、ベースレジスタのアドレスの 1 ワード後から始まります。エンコード A1 の P = 1、U = 1 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `RFE` 命令は無条件の必要があります。

`<Rn>` ベースレジスタ

`!` 命令の実行によって変更された値を `<Rn>` に書き戻すように指示します。`!` を省略した場合、命令によって `<Rn>` は変更されません。

`RFEFA`、`RFEEA`、`RFEFD`、`RFEED` は、それぞれ `RFEDA`、`RFEDB`、`RFEIA`、`RFEIB` の擬似命令で、フル上昇、空上昇、フル下降、空下降スタックからデータをポップするために使用されることを示します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !CurrentModeIsPrivileged() || CurrentInstrSet() == InstrSet_ThumbEE then
        UNPREDICTABLE;
    else
        address = if increment then R[n] else R[n]-8;
        if wordhigher then address = address+4;
        CPSRWriteByInstr(MemA[address+4,4], '1111', TRUE);
        BranchWritePC(MemA[address,4]);
        if wback then R[n] = if increment then R[n]+8 else R[n]-8;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.9 SMC (previously SMI)

Secure Monitor Call causes a Secure Monitor exception. It is available only in privileged modes. An attempt to execute this instruction in User mode causes an Undefined Instruction exception.

For details of the effects of a Secure Monitor exception see *Secure Monitor Call (SMC) exception* on page B1-53.

Encoding T1 Security Extensions (not in ARMv6K)

```
SMC<c> #<imm4>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	1	1	1	imm4				1	0	0	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
imm32 = ZeroExtend(imm4, 32);
// imm32 is for assembly/disassembly only and is ignored by hardware
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding A1 Security Extensions

```
SMC<c> #<imm4>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	1	1	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	0	1	1	1	imm4			

```
imm32 = ZeroExtend(imm4, 32);
// imm32 is for assembly/disassembly only and is ignored by hardware
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.9 SMC（以前の SMI）

SMC（セキュア モニタコール）はセキュアモニタ例外を引き起こします。特権モードでのみ使用できます。ユーザモードでこの命令の実行を試みると、未定義命令例外が発生します。

セキュアモニタ例外の影響の詳細については、*セキュア モニタコール (SMC) 例外* P. B1-53 を参照して下さい。

エンコード T1 セキュリティ拡張機能（ARMv6K には存在しない）

```
SMC<c> #<imm4>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	1	1	1	imm4				1	0	0	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

```
imm32 = ZeroExtend(imm4, 32);
// imm32 is for assembly/disassembly only and is ignored by hardware
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード A1 セキュリティ拡張機能

```
SMC<c> #<imm4>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	1	1	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	0	1	1	1	imm4			

```
imm32 = ZeroExtend(imm4, 32);
// imm32 is for assembly/disassembly only and is ignored by hardware
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SMC<c><q>  #<imm4>
```

where:

<c><q> See *Standard assembler syntax fields* on page A8-7.

<imm4> Is a 4-bit immediate value. This is ignored by the ARM processor. It can be used by the SMC exception handler (Secure Monitor code) to determine what service is being requested, but this is not recommended.

The pre-UAL syntax SMI<c> is equivalent to SMC<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if HaveSecurityExt() && CurrentModeIsPrivileged() then
        TakeSMCException();         // Secure Monitor Call if privileged
    else
        UNDEFINED;
```

Exceptions

Secure Monitor Call.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SMC<c><q>  #<imm4>
```

各項目の説明については以下を参照して下さい。

<c><q> *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

<imm4> 4 ビットのイミディエート値です。これは、ARM プロセッサによって無視されます。SMC 例外ハンドラ（セキュア モニタコード）で、要求されているサービスを特定するために使用できますが、この方法は非推奨です。

UAL 以前の構文の SMI<c> は、SMC<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if HaveSecurityExt() && CurrentModeIsPrivileged() then
        TakeSMCException();         // Secure Monitor Call if privileged
    else
        UNDEFINED;
```

例外

セキュア モニタコール

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.10 SRS

Store Return State stores the LR and SPSR of the current mode to the stack of a specified mode. For information about memory accesses see *Memory accesses* on page A8-13.

Encoding T1 ARMv6T2, ARMv7

```
SRSDB<c> SP{!},#<mode>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	0	W	0	(1)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	mode				

```
wback = (W == '1');  increment = FALSE;  wordhigher = FALSE;
// In Non-secure state, check for attempts to access Monitor mode ('10110'), or FIQ
// mode ('10001') when the Security Extensions are reserving the FIQ registers. The
// definition of UNPREDICTABLE does not permit this to be a security hole.
if !IsSecure() && mode == '10110' then UNPREDICTABLE;
if !IsSecure() && mode == '10001' && NSACR.RFR == '1' then UNPREDICTABLE;
```

Encoding T2 ARMv6T2, ARMv7

```
SRS{IA}<c> SP{!},#<mode>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	1	0	W	0	(1)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	mode				

```
wback = (W == '1');  increment = TRUE;  wordhigher = FALSE;
// In Non-secure state, check for attempts to access Monitor mode ('10110'), or FIQ
// mode ('10001') when the Security Extensions are reserving the FIQ registers. The
// definition of UNPREDICTABLE does not permit this to be a security hole.
if !IsSecure() && mode == '10110' then UNPREDICTABLE;
if !IsSecure() && mode == '10001' && NSACR.RFR == '1' then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

```
SRS{<amode>} SP{!},#<mode>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	P	U	1	W	0	(1)	(1)	(0)	(1)	(0)	(0)	(0)	(0)	(0)	(1)	(0)	(1)	(0)	(0)	(0)	mode				

```
wback = (W == '1');  inc = (U == '1');  wordhigher = (P == U);
// In Non-secure state, check for attempts to access Monitor mode ('10110'), or FIQ
// mode ('10001') when the Security Extensions are reserving the FIQ registers. The
// definition of UNPREDICTABLE does not permit this to be a security hole.
if !IsSecure() && mode == '10110' then UNPREDICTABLE;
if !IsSecure() && mode == '10001' && NSACR.RFR == '1' then UNPREDICTABLE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B6.1.10 SRS

SRS（復帰状態ストア）は、現在のモードの LR と SPSR を、指定されたモードのスタックにストアします。メモリアクセスの詳細については、*メモリアクセス* P. A8- 13 を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

```
SRSDB<c> SP{!},#<mode>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	0	W	0	(1)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	mode				

```
wback = (W == '1');  increment = FALSE;  wordhigher = FALSE;
// In Non-secure state, check for attempts to access Monitor mode ('10110'), or FIQ
// mode ('10001') when the Security Extensions are reserving the FIQ registers. The
// definition of UNPREDICTABLE does not permit this to be a security hole.
if !IsSecure() && mode == '10110' then UNPREDICTABLE;
if !IsSecure() && mode == '10001' && NSACR.RFR == '1' then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

```
SRS{IA}<c> SP{!},#<mode>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	1	0	W	0	(1)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	mode				

```
wback = (W == '1');  increment = TRUE;  wordhigher = FALSE;
// In Non-secure state, check for attempts to access Monitor mode ('10110'), or FIQ
// mode ('10001') when the Security Extensions are reserving the FIQ registers. The
// definition of UNPREDICTABLE does not permit this to be a security hole.
if !IsSecure() && mode == '10110' then UNPREDICTABLE;
if !IsSecure() && mode == '10001' && NSACR.RFR == '1' then UNPREDICTABLE;
```

エンコード A1 ARMv6*、ARMv7

```
SRS{<amode>} SP{!},#<mode>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	P	U	1	W	0	(1)	(1)	(0)	(1)	(0)	(0)	(0)	(0)	(0)	(1)	(0)	(1)	(0)	(0)	(0)	mode				

```
wback = (W == '1');  inc = (U == '1');  wordhigher = (P == U);
// In Non-secure state, check for attempts to access Monitor mode ('10110'), or FIQ
// mode ('10001') when the Security Extensions are reserving the FIQ registers. The
// definition of UNPREDICTABLE does not permit this to be a security hole.
if !IsSecure() && mode == '10110' then UNPREDICTABLE;
if !IsSecure() && mode == '10001' && NSACR.RFR == '1' then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
SRS{<amode>}<c><q>  SP{!}, #<mode>
```

where:

`<amode>` is one of:

- `DA` Decrement After. ARM code only. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0 in encoding A1.
- `DB` Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoding T1, or encoding A1 with P = 1, U = 0.
- `IA` Increment After. The consecutive memory addresses start at the address in the base register. This is the default, and is normally omitted. Encoding T2, or encoding A1 with P = 0, U = 1.
- `IB` Increment Before. ARM code only. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1 in encoding A1.

`<c><q>` See *Standard assembler syntax fields* on page A8-7. An ARM `SRS` instruction must be unconditional.

`!` Causes the instruction to write a modified value back to the base register (encoded as W = 1). If `!` is omitted, the instruction does not change the base register (encoded as W = 0).

`<mode>` The number of the mode whose banked SP is used as the base register. For details of processor modes and their numbers see *ARM processor modes* on page B1-6.

`SRSFA`, `SRSEA`, `SRSFD`, and `SRSED` are pseudo-instructions for `SRSIB`, `SRSIA`, `SRSDB`, and `SRSDA` respectively, referring to their use for pushing data onto Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentModeIsUserOrSystem() || CurrentInstrSet() == InstrSet_ThumbEE then
        UNPREDICTABLE;
    else
        base = Rmode[13,mode];
        address = if increment then base else base-8;
        if wordhigher then address = address+4;
        MemA[address,4]   = LR;
        MemA[address+4,4] = SPSR[];
        if wback then Rmode[13] = if increment then base+8 else base-8;
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アセンブラ構文

```
SRS{<amode>}<c><q>  SP{!}, #<mode>
```

各項目の説明については以下を参照して下さい。

`<amode>` 次のいずれかを指定します。

- `DA` ポストデクリメント。ARM コードのみ。連続するメモリアドレスは、ベースレジスタのアドレスで終わります。エンコード A1 の P = 0、U = 1 としてエンコードされます。
- `DB` プリデクリメント。連続するメモリアドレスは、ベースレジスタのアドレスの 1 ワード前で終わります。エンコード T1、または P = 1、U = 0 のエンコード A1。
- `IA` ポストインクリメント。連続するメモリアドレスは、ベースレジスタのアドレスで始まります。これはデフォルトで、通常は省略されます。エンコード T2、または P = 0、U = 1 のエンコード A1。
- `IB` プリインクリメント。ARM コードのみ。連続するメモリアドレスは、ベースレジスタのアドレスの 1 ワード後から始まります。エンコード A1 の P = 1、U = 1 としてエンコードされます。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。ARM `SRS` は無条件の必要があります。

`!` 命令の実行によって変更された値をベースレジスタに書き戻すように指示します (W = 1 としてエンコードされます)。`!` を省略した場合、命令によってベースレジスタは変更されません (W = 0 としてエンコードされます)。

`<mode>` バンク SP がベースレジスタとして使用されるモードの番号。プロセッサモードとそれらの番号の詳細については、*ARM プロセッサモード* P. B1-6 を参照して下さい。

`SRSFA`、`SRSEA`、`SRSFD`、`SRSED` は、それぞれ `SRSIB`、`SRSIA`、`SRSDB`、`SRSDA` の擬似命令で、フル上昇、空上昇、フル下降、空下降スタックにデータをプッシュするために使用されます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentModeIsUserOrSystem() || CurrentInstrSet() == InstrSet_ThumbEE then
        UNPREDICTABLE;
    else
        base = Rmode[13,mode];
        address = if increment then base else base-8;
        if wordhigher then address = address+4;
        MemA[address,4]   = LR;
        MemA[address+4,4] = SPSR[];
        if wback then Rmode[13] = if increment then base+8 else base-8;
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.11 STM (user registers)

Store Multiple (user registers) is UNPREDICTABLE in User or System modes. In exception modes, it stores multiple User mode registers to consecutive memory locations using an address from a banked base register. Writeback to the base register is not available with this instruction.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
STM{amode}<c> <Rn>,<registers>^
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0	P	U	1	(0)	0	Rn	register_list

```
n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

Assembler syntax

```
STM{amode}<c><q>  <Rn>, <registers>^
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`amode` is one of:

- `DA` Decrement After. The consecutive memory addresses end at the address in the base register. For this instruction, `ED`, meaning Empty Descending, is equivalent to `DA`. Encoded as P = 0, U = 0.
- `DB` Decrement Before. The consecutive memory addresses end one word below the address in the base register. For this instruction, `FD`, meaning Full Descending, is equivalent to `DB`. Encoded as P = 1, U = 0.
- `IA` Increment After. The consecutive memory addresses start at the address in the base register. This is the default, and is normally omitted. For this instruction, `EA`, meaning Empty Ascending, is equivalent to `IA`. Encoded as P = 0, U = 1.
- `IB` Increment Before. The consecutive memory addresses start one word above the address in the base register. For this instruction, `FA`, meaning Full Ascending, is equivalent to `IB`. Encoded as P = 1, U = 1.

`<Rn>` The base register. This register can be the SP.

`<registers>` Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the `STM` instruction. The registers are stored with the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address.

The pre-UAL syntax `STM<c>{amode}` is equivalent to `STM{amode}<c>`.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B6.1.11 STM（ユーザレジスタ）

STM（複数ストア）（ユーザレジスタ）は、ユーザモードまたはシステムモードでは予測不能です。例外モードでは、バンクされたベースレジスタからのアドレスを使用して、連続するメモリ位置に複数のユーザモード レジスタをストアします。この命令では、ベースレジスタへのライトバックを使用できません。

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
STM{amode}<c> <Rn>,<registers>^
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0 0	P	U	1	(0)	0	Rn	register_list

```
n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

アセンブラ構文

```
STM{amode}<c><q>  <Rn>, <registers>^
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`amode` 次のいずれかを指定します。

- `DA` ポストデクリメント。連続するメモリアドレスは、ベースレジスタのアドレスで終わります。この命令の場合は、空下降を意味する `ED` と、`DA` が等価です。P = 0、U = 0 としてエンコードされます。
- `DB` プリデクリメント。連続するメモリアドレスは、ベースレジスタのアドレスの 1 ワード前で終わります。この命令の場合は、フル下降を意味する `FD` と、`DB` が等価です。P = 1、U = 0 としてエンコードされます。
- `IA` ポストインクリメント。連続するメモリアドレスは、ベースレジスタのアドレスで始まります。これはデフォルトで、通常は省略されます。この命令の場合は、空上昇を意味する `EA` と、`IA` が等価です。P = 0、U = 1 としてエンコードされます。
- `IB` プリインクリメント。連続するメモリアドレスは、ベースレジスタのアドレスの 1 ワード後から始まります。この命令の場合は、フル上昇を意味する `FA` と、`IB` が等価です。P = 1、U = 1 としてエンコードされます。

`<Rn>` ベースレジスタ。SP を指定できます。

`<registers>` 1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストです。`STM` 命令でストアされるレジスタのセットを指定します。番号が最も小さいレジスタが最下位のメモリアドレスにストアされ、以下順にストアが行われ、番号が最も大きいレジスタが最上位のメモリアドレスにストアされます。

UAL 以前の構文の `STM<c>{amode}` は、`STM{amode}<c>` と等価です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentModeIsUserOrSystem() then UNPREDICTABLE;
    length = 4*BitCount(registers);
    address = if increment then R[n] else R[n]-length;
    if wordhigher then address = address+4;
    for i = 0 to 14
        if registers<i> == '1' then  / Store User mode ('10000') register
            MemA[address,4] = Rmode[i, '10000'];
            address = address + 4;
    if registers<15> == '1' then
        MemA[address,4] = PCStoreValue();
```

Exceptions

Data Abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentModeIsUserOrSystem() then UNPREDICTABLE;
    length = 4*BitCount(registers);
    address = if increment then R[n] else R[n]-length;
    if wordhigher then address = address+4;
    for i = 0 to 14
        if registers<i> == '1' then  / Store User mode ('10000') register
            MemA[address,4] = Rmode[i, '10000'];
            address = address + 4;
    if registers<15> == '1' then
        MemA[address,4] = PCStoreValue();
```

例外

データアボート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.12 STRBT, STRHT, and STRT

Even in privileged modes, stores to memory by these instructions are restricted in the same way as stores to memory in User mode. This is encapsulated in the `MemA_unpriv[]` and `MemU_unpriv[]` pseudocode functions. For details see *Aligned memory accesses* on page B2-31 and *Unaligned memory accesses* on page B2-32.

For details of the instructions see:

- *STRBT* on page A8-394
- *STRHT* on page A8-414
- *STRT* on page A8-416.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B6.1.12 STRBT、STRHT、STRT

これらの命令によるメモリへのストアは、特権モードでも、ユーザモードでのメモリへのストアと同様に制限されます。これは、`MemA_unpriv[]` および `MemU_unpriv[]` 擬似コード関数にカプセル化されます。詳細については、*アライン ド メモリアクセス* P. B2-31 と *アンアライン ド メモリアクセス* P. B2-32 を参照して下さい。

命令の詳細については、次の項目を参照して下さい。

- *STRBT* P. A8-394
- *STRHT* P. A8-414
- *STRT* P. A8-416

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.13 SUBS PC, LR and related instructions

The `SUBS PC, LR, #<const>` instruction provides an exception return without the use of the stack. It subtracts the immediate constant from LR, branches to the resulting address, and also copies the SPSR to the CPSR. The ARM instruction set contains similar instructions based on other data-processing operations, with a wider range of operands, or both. The use of these other instructions is deprecated, except for `MOVS PC, LR`.

Encoding T1 ARMv6T2, ARMv7

`SUBS<c> PC,LR,#<imm8>` Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	0	1	(1)	(1)	(1)	(0)	1	0	(0)	0	(1)	(1)	(1)	(1)	imm8							

```
n = 14;  imm32 = ZeroExtend(imm8, 32);  register_form = FALSE;  opcode = '0010';  // = SUB
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
<opc1>S<c> PC,<Rn>,#<const>
<opc2>S<c> PC,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	opcode				1	Rn				1	1	1	1	imm12											

```
n = UInt(Rn);  imm32 = ARMExpandImm(imm12);  register_form = FALSE;
```

Encoding A2 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
<opc1>S<c> PC,<Rn>,<Rm>{,<shift>}
<opc2>S<c> PC,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	opcode				1	Rn				1	1	1	1	imm5					type		0	Rm			

```
n = UInt(Rn);  m = UInt(Rm);  register_form = TRUE;
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

Assembler syntax

`SUBS<c><q>  PC, LR, #<const>`	Encodings T1, A1
`<opc1>S<c><q>  PC, <Rn>, #<const>`	Encoding A1
`<opc1>S<c><q>  PC, <Rn>, <Rm> {,<shift>}`	Encoding A2, deprecated
`<opc2>S<c><q>  PC, #<const>`	Encoding A1, deprecated
`<opc2>S<c><q>  PC, <Rm> {,<shift>}`	Encoding A2

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.13 SUBS PC、LR、および関連する命令

`SUBS PC, LR, #<const>` 命令は、スタックを使用しない例外からの復帰を実現します。LR からイミディエート定数を減算して、生成されたアドレスに分岐し、SPSR を CPSR にコピーします。ARM 命令セットには、その他のデータ処理操作に基づく同様の命令と、より広い範囲のオペランドを使用する同様の命令のどちらか、またはその両方が含まれます。`MOVS PC, LR` の場合を除いて、これらの他の命令の使用は非推奨です。

エンコード T1 ARMv6T2、ARMv7

`SUBS<c> PC,LR,#<imm8>` IT ブロック外部または IT ブロック内部の最後

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	0	1	(1)	(1)	(1)	(0)	1	0	(0)	0	(1)	(1)	(1)	(1)	imm8							

```
n = 14;  imm32 = ZeroExtend(imm8, 32);  register_form = FALSE;  opcode = '0010';  // = SUB
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード A1 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
<opc1>S<c> PC,<Rn>,#<const>
<opc2>S<c> PC,#<const>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	opcode				1	Rn				1	1	1	1	imm12											

```
n = UInt(Rn);  imm32 = ARMExpandImm(imm12);  register_form = FALSE;
```

エンコード A2 ARMv4*、ARMv5T*、ARMv6*、ARMv7

```
<opc1>S<c> PC,<Rn>,<Rm>{,<shift>}
<opc2>S<c> PC,<Rm>{,<shift>}
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	opcode				1	Rn				1	1	1	1	imm5					type		0	Rm			

```
n = UInt(Rn);  m = UInt(Rm);  register_form = TRUE;
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

アセンブラ構文

`SUBS<c><q>  PC, LR, #<const>`	エンコード T1、A1
`<opc1>S<c><q>  PC, <Rn>, #<const>`	エンコード A1
`<opc1>S<c><q>  PC, <Rn>, <Rm> {,<shift>}`	エンコード A2（非推奨）
`<opc2>S<c><q>  PC, #<const>`	エンコード A1（非推奨）
`<opc2>S<c><q>  PC, <Rm> {,<shift>}`	エンコード A2

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8- 7 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<opc1> The operation. <opc1> is one of ADC, ADD, AND, BIC, EOR, ORR, RSB, RSC, SBC, and SUB. Use of all of these operations except SUB is deprecated.

<opc2> The operation. <opc2> is MOV or MVN. Use of MVN is deprecated.

<Rn> The first operand register. Use of any register except LR is deprecated.

<const> The immediate constant. For encoding T1, <const> is in the range 0-255. See *Modified immediate constants in ARM instructions* on page A5-9 for the range of available values in encoding A1.

<Rm> The optionally shifted second or only operand register. Use of any register except LR is deprecated.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. The shifts and how they are encoded are described in *Shifts applied to a register* on page A8-10. Use of <shift> is deprecated.

The value of the operation <opc1> or <opc2> is encoded in the opcode field of the instruction. For the opcode values for different operations see *Operation* on page B6-6.

In Thumb code, MOVS<c><q> PC,LR is a pseudo-instruction for SUBS<c><q> PC,LR,#0.

The pre-UAL syntax <opc1><c>S is equivalent to <opc1>S<c>. The pre-UAL syntax <opc2><c>S is equivalent to <opc2>S<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet() == InstrSet_ThumbEE then
        UNPREDICTABLE;
    operand2 = if register_form then Shift(R[m], shift_t, shift_n, APSR.C) else imm32;
    case opcode of
        when '0000'  result = R[n] AND operand2;                                  // AND
        when '0001'  result = R[n] EOR operand2;                                  // EOR
        when '0010'  (result, -, -) = AddWithCarry(R[n], NOT(operand2), '1');     // SUB
        when '0011'  (result, -, -) = AddWithCarry(NOT(R[n]), operand2, '1');     // RSB
        when '0100'  (result, -, -) = AddWithCarry(R[n], operand2, '0');          // ADD
        when '0101'  (result, -, -) = AddWithCarry(R[n], operand2, APSR.c);       // ADC
        when '0110'  (result, -, -) = AddWithCarry(R[n], NOT(operand2), APSR.C);  // SBC
        when '0111'  (result, -, -) = AddWithCarry(NOT(R[n]), operand2, APSR.C);  // RSC
        when '1100'  result = R[n] OR operand2;                                   // ORR
        when '1101'  result = operand2;                                           // MOV
        when '1110'  result = R[n] AND NOT(operand2);                             // BIC
        when '1111'  result = NOT(operand2);                                      // MVN
    CPSRWriteByInstr(SPSR[], '1111', TRUE);
    BranchWritePC(result);
```

Exceptions

None.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

<opc1> 演算。<opc1> は、ADC、ADD、AND、BIC、EOR、ORR、RSB、RSC、SBC、SUB のいずれかです。SUB を除いて、これらのすべての演算の使用は非推奨です。

<opc2> 演算。<opc2> は MOV または MVN です。MVN の使用は非推奨です。

<Rn> 最初のオペランドレジスタ。LR 以外のレジスタの使用は非推奨です。

<const> イミディエート定数。エンコード T1 での <const> の範囲は 0 ～ 255 です。エンコード A1 で使用可能な値の範囲については、*ARM 命令での修飾イミディエート定数* P. A5-9 を参照して下さい。

<Rm> オプションとしてシフトされる、2 番目のオペランドレジスタまたは唯一のオペランドレジスタ。LR 以外のレジスタの使用は非推奨です。

<shift> <Rm> から読み出された値に適用されるシフト。省略した場合、シフトは適用されません。シフトと、そのエンコード方法については、*レジスタに適用されるシフト* P. A8-10 を参照して下さい。<shift> の使用は非推奨です。

演算の <opc1> または <opc2> の値は、命令の opcode フィールドにエンコードされます。各種の演算に使用される opcode 値については、*動作* P. B6-6 を参照して下さい。

Thumb コードでは、MOVS<c><q> PC,LR は、SUBS<c><q> PC,LR,#0 の擬似命令です。

UAL 以前の構文の <opc1><c>S は、<opc1>S<c> と等価です。UAL 以前の構文の <opc2><c>S は、<opc2>S<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet() == InstrSet_ThumbEE then
        UNPREDICTABLE;
    operand2 = if register_form then Shift(R[m], shift_t, shift_n, APSR.C) else imm32;
    case opcode of
        when '0000'  result = R[n] AND operand2;                                  // AND
        when '0001'  result = R[n] EOR operand2;                                  // EOR
        when '0010'  (result, -, -) = AddWithCarry(R[n], NOT(operand2), '1');     // SUB
        when '0011'  (result, -, -) = AddWithCarry(NOT(R[n]), operand2, '1');     // RSB
        when '0100'  (result, -, -) = AddWithCarry(R[n], operand2, '0');          // ADD
        when '0101'  (result, -, -) = AddWithCarry(R[n], operand2, APSR.c);       // ADC
        when '0110'  (result, -, -) = AddWithCarry(R[n], NOT(operand2), APSR.C);  // SBC
        when '0111'  (result, -, -) = AddWithCarry(NOT(R[n]), operand2, APSR.C);  // RSC
        when '1100'  result = R[n] OR operand2;                                   // ORR
        when '1101'  result = operand2;                                           // MOV
        when '1110'  result = R[n] AND NOT(operand2);                             // BIC
        when '1111'  result = NOT(operand2);                                      // MVN
    CPSRWriteByInstr(SPSR[], '1111', TRUE);
    BranchWritePC(result);
```

例外

なし

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.14 VMRS

Move to ARM core register from Advanced SIMD and VFP extension System Register moves the value of an extension system register to a general-purpose register.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD

```
VMRS<c> <Rt>,<spec_reg>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	1	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	1	1	1	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

```
t = UInt(Rt);
if t == 13 && CurrentInstrSet() != InstrSet_ARM then UNPREDICTABLE;
if t == 15 && reg != '0001' then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.14 VMRS

VMRS（アドバンスト SIMD および VFP 拡張機能システムレジスタから ARM コアレジスタへの移動）は、拡張機能システムレジスタの値を汎用レジスタに移動します。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD

```
VMRS<c> <Rt>,<spec_reg>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	1	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	1	1	1	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

```
t = UInt(Rt);
if t == 13 && CurrentInstrSet() != InstrSet_ARM then UNPREDICTABLE;
if t == 15 && reg != '0001' then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VMRS<c><q>  <Rt>, <spec_reg>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<Rt>` The destination ARM core register. This register can be R0-R14.

If `<spec_reg>` is `FPSCR`, it is also permitted to be `APSR_nzcv`, encoded as Rt = '1111'. This instruction transfers the FPSCR N, Z, C, and V flags to the APSR N, Z, C, and V flags.

`<spec_reg>` Is one of:

`FPSID`	reg = '0000'
`FPSCR`	reg = '0001'
`MVFR1`	reg = '0110'
`MVFR0`	reg = '0111'
`FPEXC`	reg = '1000'.

If the Common VFP subarchitecture is implemented, see *Subarchitecture additions to the VFP system registers* on page AppxB-15 for additional values of `<spec_reg>`.

The pre-UAL instruction `FMSTAT` is equivalent to `VMRS APSR_nzcv, FPSCR`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then // FPSCR
        CheckVFPEnabled(TRUE);  SerializeVFP();  VFPExcBarrier();
        if t == 15 then
            APSR.N = FPSCR.N;  APSR.Z = FPSCR.Z;  APSR.C = FPSCR.C;  APSR.V = FPSCR.V;
        else
            R[t] = FPSCR;
    else // Non-FPSCR registers are privileged-only and not affected by FPEXC.EN
        CheckVFPEnabled(FALSE);
        if !CurrentModeIsPrivileged() then UNDEFINED;
        case reg of
            when '0000'  SerializeVFP();  R[t] = FPSID;
            // '0001' already dealt with above
            when '001x'  UNPREDICTABLE;
            when '010x'  UNPREDICTABLE;
            when '0110'  SerializeVFP();  R[t] = MVFR1;
            when '0111'  SerializeVFP();  R[t] = MVFR0;
            when '1000'  SerializeVFP();  R[t] = FPEXC;
            otherwise    SUBARCHITECTURE_DEFINED register access;
```

Exceptions

Undefined Instruction.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
VMRS<c><q>  <Rt>, <spec_reg>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<Rt>` デスティネーションの ARM コアレジスタ。このレジスタには、R0 ～ R14 を指定できます。

`<spec_reg>` が `FPSCR` の場合は、Rt = '1111' としてエンコードされる `APSR_nzcv` も指定できます。この命令は、FPSCR の N、Z、C、V フラグを APSR の N、Z、C、V フラグに転送します。

`<spec_reg>` 次のいずれかを指定します。

`FPSID`.	reg = '0000'
`FPSCR`.	reg = '0000'
`MVFR1`.	reg = '0110'
`MVFR0`.	reg = '0111'
`FPEXC`.	reg = '1000'

共通 VFP サブアーキテクチャが実装されている場合は、`<spec_reg>` のその他の値について *VFP システムレジスタへのサブアーキテクチャ固有の追加* P. 付録 B-15 を参照して下さい。

UAL 以前の命令の `FMSTAT` は、`VMRS APSR_nzcv, FPSCR` と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then // FPSCR
        CheckVFPEnabled(TRUE);  SerializeVFP();  VFPExcBarrier();
        if t == 15 then
            APSR.N = FPSCR.N;  APSR.Z = FPSCR.Z;  APSR.C = FPSCR.C;  APSR.V = FPSCR.V;
        else
            R[t] = FPSCR;
    else // Non-FPSCR registers are privileged-only and not affected by FPEXC.EN
        CheckVFPEnabled(FALSE);
        if !CurrentModeIsPrivileged() then UNDEFINED;
        case reg of
            when '0000'  SerializeVFP();  R[t] = FPSID;
            // '0001' already dealt with above
            when '001x'  UNPREDICTABLE;
            when '010x'  UNPREDICTABLE;
            when '0110'  SerializeVFP();  R[t] = MVFR1;
            when '0111'  SerializeVFP();  R[t] = MVFR0;
            when '1000'  SerializeVFP();  R[t] = FPEXC;
            otherwise    SUBARCHITECTURE_DEFINED register access;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.15 VMSR

Move to Advanced SIMD and VFP extension System Register from ARM core register moves the value of a general-purpose register to a VFP system register.

Encoding T1 / A1 VFPv2, VFPv3, Advanced SIMD

```
VMSR<c> <spec_reg>,<Rt>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	0	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	1	1	0	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

```
t = UInt(Rt);
if t == 15 || (t == 13 && CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B6.1.15 VMSR

VMSR（ARM コアレジスタからアドバンスト SIMD および VFP 拡張機能システムレジスタへの移動）は、汎用レジスタの値を VFP システムレジスタに移動します。

エンコード T1/A1 VFPv2、VFPv3、アドバンスト SIMD

```
VMSR<c> <spec_reg>,<Rt>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	0	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	1	1	0	1	1	1	0	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

```
t = UInt(Rt);
if t == 15 || (t == 13 && CurrentInstrSet() != InstrSet_ARM) then UNPREDICTABLE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Assembler syntax

```
VMSR<c><q>  <spec_reg>, <Rt>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A8-7.

`<spec_reg>` Is one of:

FPSID	reg = '0000'
FPSCR	reg = '0001'
FPEXC	reg = '1000'.

If the Common VFP subarchitecture is implemented, see *Subarchitecture additions to the VFP system registers* on page AppxB-15 for additional values of `<spec_reg>`.

`<Rt>` The general-purpose register to be transferred to `<spec_reg>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then // FPSCR
        CheckVFPEnabled(TRUE);  SerializeVFP();  VFPExcBarrier();
        FPSCR = R[t];
    else // Non-FPSCR registers are privileged-only and not affected by FPEXC.EN
        CheckVFPEnabled(FALSE);
        if !CurrentModeIsPrivileged() then UNDEFINED;
        case reg of
            when '0000'  SerializeVFP();
            // '0001' already dealt with above
            when '001x'  UNPREDICTABLE;
            when '01xx'  UNPREDICTABLE;
            when '1000'  SerializeVFP();  FPEXC = R[t];
            otherwise    SUBARCHITECTURE_DEFINED register access;
```

Exceptions

Undefined Instruction.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アセンブラ構文

```
VMSR<c><q>  <spec_reg>, <Rt>
```

各項目の説明については以下を参照して下さい。

`<c><q>` *標準アセンブラ構文のフィールド* P. A8-7 を参照して下さい。

`<spec_reg>` 次のいずれかを指定します。

FPSID	reg = '0000'
FPSCR	reg = '0000'
FPEXC	reg = '1000'

共通 VFP サブアーキテクチャが実装されている場合は、`<spec_reg>` のその他の値について *VFP システムレジスタへのサブアーキテクチャ固有の追加* P. 付録 B-15 を参照して下さい。

`<Rt>` `<spec_reg>` に転送される汎用レジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then // FPSCR
        CheckVFPEnabled(TRUE);  SerializeVFP();  VFPExcBarrier();
        FPSCR = R[t];
    else // Non-FPSCR registers are privileged-only and not affected by FPEXC.EN
        CheckVFPEnabled(FALSE);
        if !CurrentModeIsPrivileged() then UNDEFINED;
        case reg of
            when '0000'  SerializeVFP();
            // '0001' already dealt with above
            when '001x'  UNPREDICTABLE;
            when '01xx'  UNPREDICTABLE;
            when '1000'  SerializeVFP();  FPEXC = R[t];
            otherwise    SUBARCHITECTURE_DEFINED register access;
```

例外

未定義命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Part C

Debug Architecture

抄訳

パート C

デバッグアーキテクチャ

Chapter C1
Introduction to the ARM Debug Architecture

This chapter introduces part C of this manual, and the ARM Debug architecture. It contains the following sections:

- *Scope of part C of this manual* on page C1-2
- *About the ARM Debug architecture* on page C1-3
- *Security Extensions and debug* on page C1-8
- *Register interfaces* on page C1-9.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 C1 章
ARM デバッグアーキテクチャの概要

本章では、本書のパート C で解説する ARM デバッグアーキテクチャの概要を紹介します。本章は次のセクションから構成されています。

- *本書のパート C の範囲* P. C1-2
- *ARM デバッグアーキテクチャについて* P. C1-3
- *セキュリティ拡張機能とデバッグ* P. C1-8
- *レジスタインタフェース* P. C1-9

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C1.1 Scope of part C of this manual

Part C of this manual defines the debug features of ARMv7. However, ARM recognizes that many debuggers require compatibility with previous versions of the ARM Debug architecture. Therefore, this part includes information about three versions of the ARM Debug architecture:

- v7 Debug
- v6.1 Debug
- v6 Debug

These three versions of the Debug architecture are introduced in *Major differences between the ARMv6 and ARMv7 Debug architectures* on page C1-7.

In part C of this manual:

- ARMv6 is used sometimes to refer to an implementation that includes either v6.1 Debug or v6 Debug.
- ARMv7 is used sometimes to refer to an implementation that includes v7 Debug.

Note

- v6.1 Debug and v6 Debug are two different versions of the Debug architecture for the ARMv6 architecture. They might be described as:
  - — ARMv6, v6.1 Debug
  - — ARMv6, v6 Debug.

  Throughout this part the descriptions v6.1 Debug and v6 Debug are used, for brevity.
- Any processor that implements the ARMv7 architecture must implement v7 Debug. Information about v6.1 Debug and v6 Debug is given:
  - — to enable developers to produce debuggers that are backwards compatible with these Debug architecture versions
  - — as reference material for processors that implement the ARMv6 architecture.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C1.1 本書のパート C の範囲

本書のパート C では、ARMv7 のデバッグ機能を定義します。ただし、多くのデバッガでは、ARM デバッグアーキテクチャの従来のバージョンとの互換性が必要です。このため、パート C では ARM デバッグアーキテクチャの、次に示す 3 つのバージョンについて説明します。

- v7 デバッグ
- v6.1 デバッグ
- v6 デバッグ

デバッグアーキテクチャのこれら 3 つのバージョンについては、*ARMv6 と ARMv7 とのデバッグアーキテクチャの主な相違点* P. C1-7 で概説されています。

本書のパート C では、用語を次の意味で使用します。

- ARMv6 という用語は、v6.1 デバッグまたは v6 デバッグを含む実装を指すために使用されることがあります。
- ARMv7 という用語は、v7 デバッグを含む実装を指すために使用されることがあります。

注

- v6.1 デバッグと v6 デバッグは、ARMv6 アーキテクチャの、デバッグアーキテクチャにおける 2 つの異なるバージョンです。これらは、次のように記述される場合があります。
  - – ARMv6、v6.1 デバッグ
  - – ARMv6、v6 デバッグ

  記述を簡潔にするため、パート C を通じて v6.1 デバッグおよび v6 デバッグという表記を使用します。
- ARMv7 アーキテクチャを実装するプロセッサは、必ず v7 デバッグを実装する必要があります。v6.1 デバッグおよび v6 デバッグについての情報は、次の目的で提供されます。
  - – これらのデバッグアーキテクチャのバージョンと下位互換性を持つデバッガを開発する開発者のため
  - – ARMv6 アーキテクチャを実装するプロセッサ向けの参照資料として

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C1.2 About the ARM Debug architecture

ARM processors implement two types of debug support:

Invasive debug All debug features that permit modification of processor state. For more information, see *Invasive debug*.

Non-invasive debug All debug features that permit data and program flow observation, especially trace support. For more information, see *Non-invasive debug* on page C1-5.

The following sections introduce invasive and non-invasive debug. *Summary of the ARM debug component descriptions* on page C1-7 gives a quick reference summary of the rest of this part of this manual.

C1.2.1 Invasive debug

The invasive debug component of the ARM Debug architecture is intended primarily for run-control debugging.

Note

In this part of this manual, invasive debug is often referred to simply as debug. For example, debug events, debug exceptions, and Debug state are all part of the invasive debug implementation.

The programmers' model can be used to manage and control *debug events*. Watchpoints and breakpoints are two examples of debug events. Debug events are described in Chapter C3 *Debug Events*.

You can configure the processor through the DBGDSCR into one of two debug-modes:

Monitor debug-mode

In Monitor debug-mode, a debug event causes a *debug exception* to occur:

- a debug exception that relates to instruction execution generates a Prefetch Abort exception
- a debug exception that relates to a data access generates a Data Abort exception.

Debug exceptions are described in Chapter C4 *Debug Exceptions*.

Halting debug-mode

In Halting debug-mode, a debug event causes the processor to enter a special *Debug state*. When the processor is in Debug state, the processor ceases to execute instructions from the program counter location, but is instead controlled through the external debug interface, in particular the *Instruction Transfer Register* (DBGITR). This enables an external agent, such as a debugger, to interrogate processor context, and control all subsequent instruction execution. Because the processor is stopped, it ignores the system and cannot service interrupts.

Debug state is described in Chapter C5 *Debug State*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C1.2 ARM デバッグアーキテクチャについて

ARM プロセッサには、次の 2 種類のデバッグサポートが実装されます。

侵襲性デバッグ プロセッサ状態の変更を許可する、すべてのデバッグ機能。詳細については、*侵襲性デバッグ*を参照して下さい。

非侵襲性デバッグ データおよびプログラムフローの観測を許可するすべてのデバッグ機能、特にトレースサポート機能。詳細については、*非侵襲性デバッグ* P. C1-5 を参照して下さい。

以下のセクションでは、侵襲性および非侵襲性のデバッグについて説明します。本書のパート C のクイックリファレンスを、*ARM デバッグコンポーネントの説明の要約* P. C1-7 に示します。

C1.2.1 侵襲性デバッグ

ARM デバッグアーキテクチャの侵襲性デバッグコンポーネントは、主に実行制御デバッグを目的としています。

注

本書のパート C では、多くの場合侵襲性デバッグを単にデバッグと記述します。例えば、デバッグイベント、デバッグ例外、デバッグ状態はすべて、侵襲性デバッグの実装の一部です。

プログラマモデルを使用して、*デバッグイベント*の管理と制御を実行できます。ウォッチポイントとブレークポイントは、デバッグイベントの 2 つの例です。デバッグイベントについては、第 C3 章*「デバッグイベント」*を参照して下さい。

DBGDSCR を使用して、プロセッサを 2 つのデバッグモードのいずれかに構成できます。

モニタ デバッグモード

モニタ デバッグモードでは、デバッグイベントにより*デバッグ例外*が発生します。

- 命令の実行に関連するデバッグ例外は、プリフェッチアボート例外を生成します。
- データアクセスに関連するデバッグ例外は、データアボート例外を生成します。

デバッグ例外については、第 C4 章*「デバッグ例外」*を参照して下さい。

ホールト デバッグモード

ホールト デバッグモードでは、デバッグイベントが発生すると、プロセッサが特別な*デバッグ状態*に入ります。プロセッサがデバッグ状態であるとき、プロセッサはプログラムカウンタ位置からの命令を実行せず、外部のデバッグインタフェース、特に*命令転送レジスタ*(DBGITR) により制御されます。これによって、デバッガなど外部のエージェントがプロセッサのコンテキストを調査し、以後の命令の実行をすべて制御できます。プロセッサは停止しており、システムを無視するため、割り込みを処理できません。

デバッグ状態については、第 C5 章*「デバッグ状態」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A debug solution can use a mixture of the two methods, for example to support an OS or RTOS with both:

- *Running System Debug* (RSD) using Monitor debug-mode
- Halting debug-mode support available as a fallback for system failure and boot time debug.

The architecture supports the ability to switch between these two debug-modes.

When no debug-mode is selected, debug is restricted to simple monitor solutions. These are usually ROM or Flash-based. Such a monitor might use standard system features, such as a UART or Ethernet connection, to communicate with a debug host. Alternatively, it might use the *Debug Communications Channel* (DCC) as an out-of-band communications channel to the host. This minimizes the debug requirement on system resources.

All versions of the Debug architecture provide a software interface that includes:

- a *Debug Identification Register* (DBGDIDR)
- status and control registers, including the *Debug Status and Control Register* (DBGDSCR)
- hardware breakpoint and watchpoint support
- the DCC.

In addition, the v7 Debug software interface includes reset, power-down and operating system debug support features.

The Debug architecture requires an external debug interface that supports access to the programmers' model.

This forms the basis of the *Debug Programmers' Model* (DPM) for ARMv6 and ARMv7.

Description of invasive debug features

The following chapters describe the invasive debug implementation:

- Chapter C2 *Invasive Debug Authentication*
- Chapter C3 *Debug Events*
- Chapter C4 *Debug Exceptions*
- Chapter C5 *Debug State*.

In addition, see:

- Chapter C6 *Debug Register Interfaces* for a description of the register interfaces to the debug components
- Chapter C10 *Debug Registers Reference* for descriptions of the registers used to configure and control debug operations
- Appendix A *Recommended External Debug Interface* for a description of the recommended external interface to the debug components.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

デバッグソリューションでは 2 つの手法を混在させることができます。例えば、OS や RTOS で次の両方をサポートできます。

- *実行中システムデバッグ*(RSD) では、モニタ デバッグモードを使用します。
- システム障害時の代替機構およびブート時のデバッグとして、ホールト デバッグモードをサポートします。

アーキテクチャは、これら 2 つのデバッグモードを切り替える機能をサポートしています。

デバッグモードが選択されていない場合、デバッグは単純なモニタソリューションに制限されます。これらは通常、ROM またはフラッシュメモリをベースとしたものです。このようなモニタは、UART や Ethernet 接続などの標準的なシステム機能を使用して、デバッグホストと通信できます。または、ホストへの特別な通信チャネルとして*デバッグ通信チャネル*(DCC) を使用することもできます。これにより、デバッグに必要なシステムリソースの要件が最小になります。

すべてのバージョンのデバッグアーキテクチャでは、次の要素を含むソフトウェア インタフェースが提供されています。

- *デバッグ識別レジスタ*(DBGDIDR)
- *デバッグステータスおよび制御レジスタ*(DBGDSCR) を含む、ステータスおよび制御レジスタ
- ハードウェアのブレークポイントおよびウォッチポイントのサポート
- DCC

さらに、v7 デバッグ ソフトウェアインタフェースには、リセット、電力オフ、およびオペレーティングシステムのデバッグサポート機能が含まれています。

デバッグアーキテクチャには、プログラマモデルへのアクセスをサポートする外部デバッグインタフェースが必要です。

これらの要素により、ARMv6 および ARMv7 の *デバッグ プログラマモデル*(DPM) の基礎が形成されます。

侵襲性デバッグ機能の説明

次に示す章では、侵襲性デバッグの実装について説明します。

- 第 C2 章 *「侵襲性デバッグの認証」*
- 第 C3 章 *「デバッグイベント」*
- 第 C4 章 *「デバッグ例外」*
- 第 C5 章 *「デバッグ状態」*

その他、次の章を参照して下さい。

- デバッグコンポーネントへのレジスタインタフェースの説明については、第 C6 章 *「デバッグレジスタ インタフェース」*を参照して下さい。
- デバッグ動作の構成と制御に使用されるレジスタの説明については、第 C10 章 *「デバッグレジスタのインタフェース」*を参照して下さい。
- 付録 A *「推奨される外部デバッグインタフェース」*デバッグコンポーネントへの外部インタフェースに関する推奨構成の説明については、を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C1.2.2 Non-invasive debug

Non-invasive debug includes all debug features that permit data and program flow to be observed, but that do not permit modification of the main processor state.

The v7 Debug architecture defines three areas of non-invasive debug:

- Instruction trace and, in some implementations, data trace. Trace support is an architecture extension typically implemented using a trace macrocell, see *Trace*.
- Sample-based profiling, see *Sample-based profiling* on page C1-6.
- Performance monitors, see *Performance monitors* on page C1-6.

A processor implementation might include other forms of non-invasive debug.

Chapter C7 *Non-invasive Debug Authentication* describes the authentication of non-invasive debug operations.

Trace

Trace support is an architecture extension. This manual describes such an extension as a trace macrocell. A trace macrocell constructs a real-time trace stream corresponding to the operation of the processor. It is IMPLEMENTATION DEFINED whether the trace stream is:

- stored locally in an *Embedded Trace Buffer* (ETB) for independent download and analysis
- exported directly through a trace port to a *Trace Port Analyzer* (TPA) and its associated host based trace debug tools.

Typically, use of a trace macrocell is non-invasive. Development tools can connect to the trace macrocell, configure it, capture trace and download the trace without affecting the operation of the processor in any way. A trace macrocell provides an enhanced level of runtime system observation and debug granularity. It is particularly useful in cases where:

- Stopping the processor affects the behavior of the system.
- There is insufficient state visible in a system by the time a problem is detected to be able to determine its cause. Trace provides a mechanism for system logging and back tracing of faults.

Trace might also be used to perform analysis of code running on the processor, such as performance analysis or code coverage analysis.

Typically, a trace architecture defines:

- the trace macrocell programmers' model
- permitted trace protocol formats
- the physical trace port connector.

The following documents define the ARM trace architectures:

- *Embedded Trace Macrocell Architecture Specification*
- *CoreSight Program Flow Trace Architecture Specification.*

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C1.2.2 非侵襲性デバッグ

非侵襲性デバッグには、データとプログラムのフローの観測は許可し、メインプロセッサの状態の変更は許可しない、すべてのデバッグ機能が含まれます。

v7 デバッグアーキテクチャでは、非侵襲性デバッグとして次の 3 つの分野が定義されています。

- 命令トレース、および一部の実装ではデータトレース。トレースサポートは、一般にトレースマクロセルを使用して実装されるアーキテクチャ拡張機能です。*トレース*を参照して下さい。
- サンプルベースのプロファイリング。*サンプルベースのプロファイリング* P. C1-6 を参照して下さい。
- パフォーマンスモニタ。*パフォーマンスモニタ* P. C1-6 を参照して下さい。

プロセッサの実装には、他の形式の非侵襲性デバッグを含めることもできます。

非侵襲性デバッグ動作の認証については、第 C7 章*「非侵襲性デバッグの認証」*を参照して下さい。

トレース

トレースサポートは、アーキテクチャの拡張機能です。本書では、このような拡張機能をトレースマクロセルとして説明します。トレースマクロセルは、プロセッサの動作に対応するリアルタイムのトレースストリームを構築します。トレースストリームについて、次の構成は実装定義です。

- *エンベデッドトレース バッファ*(ETB) にローカルに格納され、独立してダウンロードと解析が可能であるかどうか。
- トレースポート経由で、*トレースポート アナライザ*(TPA) および関連するホストベースのトレースデバッグ ツールへ直接エクスポートされるかどうか。

一般に、トレースマクロセルの使用は非侵襲性です。開発ツールでは、プロセッサの動作にどのような影響も及ぼすことなく、トレースマクロセルへの接続と構成、トレースのキャプチャとダウンロードを実行できます。トレースマクロセルを使用すると、実行中のシステムの観測およびデバッグの粒度が拡張されます。この機能は、次のような条件で特に有用です。

- プロセッサを停止すると、システムの動作に影響が及ぶ場合。
- 問題が検出された時点で、原因を判定するために十分なシステム状態が可視でない場合トレースにより、システムのログ出力を行い、フォールトのバックトレースが可能になります。

トレースは、プロセッサで実行されているコードの分析、例えばパフォーマンス分析やコードのカバー率分析を行うためにも使用できます。

一般に、トレースアーキテクチャは次のものを定義します。

- トレースマクロセルのプログラマモデル
- 許可されるトレースプロトコルのフォーマット
- 物理トレースポート コネクタ

ARM トレースアーキテクチャの定義については、次のドキュメントを参照して下さい。

- *エンベデッドトレースマクロセル アーキテクチャ仕様*
- *CoreSight プログラムフロー トレースアーキテクチャ仕様*

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The ARM trace architectures have a common identification mechanism. This means development tools can detect which architecture is implemented.

Sample-based profiling

Sample-based profiling is an optional non-invasive component of the Debug architecture, that enables debug software to profile a program. For more information, see Chapter C8 *Sample-based Profiling*.

Performance monitors

Performance monitors were implemented in several processors before ARMv7, but before ARMv7 they did not form part of the architecture. The ARMv7 form of the monitors, described here, follows those implementations with minor modifications to enable future expansion.

The basic form of the performance monitors is:

- A cycle counter, with the ability to count every cycle or every sixty-fourth cycle.
- A number of event counters. The event counted by each counter is programmable:
  - — Previous implementations provided up to four counters
  - — In ARMv7, space is provided for up to 31 counters. The actual number of counters is IMPLEMENTATION DEFINED, and an identification mechanism is provided.
- Controls for
  - — enabling and resetting counters
  - — flagging overflows
  - — enabling interrupts on overflow.

  The cycle counter can be enabled independently from the event counters.

The set of events that can be monitored is divided into:

- events that are likely to be consistent across many microarchitectures
- other events, that are likely to be implementation specific.

As a result, the architecture defines a common set of events to be used across many microarchitectures, and a large space reserved for IMPLEMENTATION DEFINED events. The full set of events for any given implementation is IMPLEMENTATION DEFINED. There is no requirement to implement any of the common set of events, but the numbers allocated for the common set of events must not be used except as defined.

Chapter C9 *Performance Monitors* describes the performance monitors.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ARM トレースアーキテクチャには、共通の識別機構が存在します。これによって、開発ツールは実装されているアーキテクチャを検出できます。

サンプルベースのプロファイリング

サンプルベースのプロファイリングは、デバッグアーキテクチャのオプションの非侵襲性コンポーネントで、デバッグソフトウェアがプログラムのプロファイリングを実行できるようにします。詳細については、第 C8 章*「サンプルベースのプロファイリング」*を参照して下さい。

パフォーマンスモニタ

パフォーマンスモニタは、ARMv7 以前のいくつかのプロセッサに実装されていましたが、ARMv7 以前はアーキテクチャの一部ではありませんでした。ARMv7 形式のモニタは、ここで説明するように、それらの実装に従いながら、将来の拡張を可能にするため小さな変更が加えられています。

パフォーマンスモニタの基本形式は次のとおりです。

- サイクルカウンタ。すべてのサイクル、または 64 サイクルごとのカウント機能があります。
- 一連のイベントカウンタ。各カウンタで、どのイベントをカウントするかをプログラム可能です。
  - – 以前の実装では、4 つまでのカウンタが提供されます。
  - – ARMv7 では、最大 31 個のカウンタの容量が提供されています。カウンタの実際の数は実装定義で、識別機構が用意されています。
- 次の動作を行うコントロール
  - – カウンタの許可とリセット
  - – オーバフローのフラグ付け
  - – オーバフロー時の割り込み可能

  サイクルカウンタは、イベントカウンタとは独立して稼動できます。

監視対象とできるイベントの組は、次のように分割されます。

- 多くのマイクロアーキテクチャについて一貫している可能性が高いイベント
- 実装固有の可能性が高い、他のイベント

結果として、アーキテクチャでは、多くのマイクロアーキテクチャにわたって使用される共通のイベントの組と、実装定義イベント用に予約される大きな容量が定義されます。特定の実装について、イベントの完全な組は実装定義です。共通のイベントの組を実装することは必須ではありません。ただし、共通のイベントの組用に割り当てられている番号は、定義以外の内容で使用することはできません。

パフォーマンスモニタについては、第 C9 章*「パフォーマンスモニタ」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C1.2.3 Major differences between the ARMv6 and ARMv7 Debug architectures

ARMv6 is the first version of the ARM architecture to include debug. The introduction of the ARM architecture Security Extensions extended the ARMv6 Debug architecture:

- ARMv6 processors without the Security Extensions implement v6 Debug
- ARMv6 processors with the Security Extensions implement v6.1 Debug.

ARMv7 introduces additional extensions to support developments in the debug environment.

The main change in the Debug architecture is the specification of new forms of external debug interface. ARMv6 Debug does not require a particular debug interface, but can be implemented with access from a JTAG interface as defined in *IEEE Standard Test Access Port and Boundary Scan Architecture (JTAG)*. However, systems such as the ARM CoreSight™ architecture require changes in the debug interface. For more information about the CoreSight architecture see the *CoreSight Architecture Specification*. ARMv7 Debug addresses some of the aims of the CoreSight architecture, such as a more system-centric view of debug, and improved debug of powered-down systems.

v7 Debug also introduces an architecture extension to provide performance monitors.

C1.2.4 Summary of the ARM debug component descriptions

Table C1-1 shows the main components of v7 Debug, and where they are described.

Table C1-1 v7 Debug subarchitectures

Component	Status	Type	Reference
Run-control Debug	Required	Invasive	Chapter C2 *Invasive Debug Authentication*
			Chapter C3 *Debug Events*
			Chapter C4 *Debug Exceptions*
			Chapter C5 *Debug State*
			Chapter C6 *Debug Register Interfaces*
Trace	Optional	Non-invasive[a]	*Trace* on page C1-5
Sample-based profiling	Optional	Non-invasive[a]	Chapter C8 *Sample-based Profiling*
Performance monitors	Optional	Non-invasive[a]	Chapter C9 *Performance Monitors*

a. For information about authentication of these components see Chapter C7 *Non-invasive Debug Authentication*.

For more information, see:

- Chapter C10 *Debug Registers Reference*
- Appendix A *Recommended External Debug Interface*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C1.2.3 ARMv6 と ARMv7 とのデバッグアーキテクチャの主な相違点

ARMv6 は、デバッグ機能が含まれている最初の ARM アーキテクチャのバージョンです。ARM アーキテクチャにセキュリティ拡張機能が導入されたため、ARMv6 のデバッグアーキテクチャは次のように拡張されています。

- セキュリティ拡張機能がない ARMv6 プロセッサは、v6 デバッグを実装しています。
- セキュリティ拡張機能がある ARMv6 プロセッサは、v6.1 デバッグを実装しています。

ARMv7 では、デバッグ環境での開発をサポートするため、新たな拡張機能が導入されています。

デバッグアーキテクチャの主な変更点は、新しい形式の外部デバッグインタフェース仕様です。ARMv6 デバッグでは特定のデバッグインタフェースは要求されませんが、*IEEE 標準のテストアクセスポートとバウンダリスキャン アーキテクチャ*(JTAG) で定義されている JTAG インタフェースからのアクセスを実装することができます。しかし、ARM CoreSight ™アーキテクチャなどのシステムでは、デバッグインタフェースの変更が必要です。CoreSight アーキテクチャの詳細については、『*CoreSight アーキテクチャ仕様*』を参照して下さい。ARMv7 のデバッグは、よりシステム寄りの視点でのデバッグなど、CoreSight アーキテクチャの目的のいくつかに対応しており、電力がオフになっているシステムのデバッグ機能が強化されています。

v7 デバッグでは、パフォーマンスモニタを提供するためのアーキテクチャ拡張機能も導入されています。

C1.2.4 ARM デバッグコンポーネントの説明の要約

v7 デバッグの主要なコンポーネントと、それぞれについて説明されている章を、表 C1-1 に示します。

表 C1-1 v7 デバッグのサブアーキテクチャ

コンポーネント	ステータス	タイプ	参照先
実行制御デバッグ	必須	侵襲性	第 C2 章*「侵襲性デバッグの認証」*
			第 C3 章*「デバッグイベント」*
			第 C4 章*「デバッグ例外」*
			第 C5 章*「デバッグ状態」*
			第 C6 章*「デバッグレジスタ インタフェース」*
トレース	オプション	非侵襲性[a]	*トレース* P. C1-5
サンプルベースのプロファイリング	オプション	非侵襲性[a]	第 C8 章*「サンプルベースのプロファイリング」*
パフォーマンスモニタ	オプション	非侵襲性[a]	第 C9 章*「パフォーマンスモニタ」*

a. これらのコンポーネントの認証については、第 C7 章*「非侵襲性デバッグの認証」*を参照して下さい。

詳細については、次の各章を参照して下さい。

- 第 C10 章*「デバッグレジスタのインタフェース」*
- 付録 A*「推奨される外部デバッグインタフェース」*

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C1.3 Security Extensions and debug

Security Extensions debug enables you to *not permit* invasive debug events and non-invasive debug operations independently in either:

- In all processor modes in Secure state.
- In Secure privileged modes but not in Secure User mode. In v7 Debug, for invasive debug events that cause entry to Debug state:
  - — support for not permitting these events is optional
  - — if an implementation does support not permitting these events the use of them is deprecated.

This is controlled by two control bits in the Secure Debug Enable Register and, in the recommended external debug interface, four input signals:

- the *Secure User Invasive Debug Enable* bit, SDER.SUIDEN
- the *Secure User Non-invasive Debug Enable* bit, SDER.SUNIDEN
- in the recommended external debug interface:
  - — the *Debug Enable* signal, **DBGEN**
  - — the *Non-Invasive Debug Enable* signal, **NIDEN**
  - — the *Secure Privileged Invasive Debug Enable* signal, **SPIDEN**
  - — the *Secure Privileged Non-Invasive Debug Enable* signal, **SPNIDEN**.

For more information, see:

- Chapter C2 *Invasive Debug Authentication*
- Chapter C7 *Non-invasive Debug Authentication*
- *c1, Secure Debug Enable Register (SDER)* on page B3-108 for details of the SUIDEN and SUNIDEN control bits
- *Authentication signals* on page AppxA-3 for details of the **DBGEN**, **NIDEN**, **SPIDEN** and **SPNIDEN** signals.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C1.3 セキュリティ拡張機能とデバッグ

セキュリティ拡張機能デバッグでは、次のいずれかの条件について、侵襲性デバッグイベントと非侵襲性デバッグイベントの動作を別々に*禁止*することができます。

- セキュア状態のすべてのプロセッサモード
- セキュア特権モードで、セキュア ユーザモードではない場合。v7 デバッグでは、デバッグ状態への遷移を引き起こす侵襲性デバッグイベントについて、次の条件があります。
  - – これらのイベントを禁止する動作のサポートはオプションです。
  - – 実装でこれらのイベントの禁止をサポートしている場合、それらのイベントの使用は推奨されません。

この機能は、セキュアデバッグ イネーブルレジスタの 2 つの制御ビット、および推奨される外部デバッグインタフェースでは次の 4 つの入力信号により制御されます。

- *セキュアユーザ侵襲性デバッグイネーブル*ビット、SDER.SUIDEN
- *セキュアユーザ非侵襲性デバッグイネーブル*ビット、SDER.SUNIDEN
- 推奨される外部デバッグインタフェースの次の信号
  - – *デバッグイネーブル*信号、DBGEN
  - – *非侵襲性デバッグイネーブル*信号、NIDEN
  - – *セキュア特権侵襲性デバッグイネーブル*信号、SPIDEN
  - – *セキュア特権非侵襲性デバッグイネーブル*信号、SPNIDEN

詳細については、次の章を参照して下さい。

- 第 C2 章*「侵襲性デバッグの認証」*
- 第 C7 章*「非侵襲性デバッグの認証」*
- SUIDEN および SUNIDEN 制御ビットの詳細については、*c1、セキュアデバッグ イネーブルレジスタ (SDER)* P. B3-108 を参照して下さい。
- DBGEN、NIDEN、SPIDEN、SPNIDEN 信号の詳細については、*認証信号* P. 付録 A-3 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C1.4 Register interfaces

This section gives a brief description of the different debug register interfaces defined by v7 Debug. The most important distinction is between:

- the external debug interface, that defines how an external debugger can access the v7 Debug resources
- the processor interface, that describes how an ARMv7 processor can access its own debug resources.

For v7 Debug, ARM recommends an external debug interface based on the *ARM Debug Interface v5 Architecture Specification* (ADIv5). The most significant difference between ADIv5 and the interface recommended by v6 Debug and v6.1 Debug is that ADIv5 supports debug over power-down of the processor.

Although the ADIv5 interface is not required for compliance with ARMv7, the ARM RealView® tools require this interface to be implemented.

ADIv5 supports both a JTAG wire interface and a low pin-count *Serial Wire* (SW) interface. The RealView tools support either wire interface.

An ADIv5 interface enables a debug object, such as an ARMv7 processor, to abstract a set of resources as a memory-mapped peripheral. Accesses to debug resources are made as 32-bit read/write transfers. Power-down debug is supported by introducing the abstraction that accesses to certain resources can return an error response when they are unavailable, just as a memory-mapped peripheral can return a slave-generated error response in exceptional circumstances.

v7 Debug requires software executing on the processor to be able to access all debug registers. To provide access to a particular basic subset of debug registers, v7 Debug requires implementation of the Baseline Coprocessor 14 (CP14) Interface, see *The Baseline CP14 debug register interface* on page C6-32. To provide access to the rest of the debug registers v7 Debug permits one of two options:

- An Extended CP14 interface. This is similar to the requirement of v6 Debug and v6.1 Debug.
- A memory-mapped interface.

An implementation can include both of these options.

ARMv7 does not permit all combinations of debug, trace, and performance monitor register interfaces. There are three options for ARMv7 implementations, shown in Table C1-2 on page C1-10. In a number of cases an optional memory-mapped interface is permitted, indicated by brackets. ARM recommends that if the optional memory-mapped interface is implemented for either the debug interface or the trace interface then it is implemented for both of these interfaces.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C1.4 レジスタインタフェース

このセクションでは、v7 デバッグで定義されている各種のデバッグレジスタ インタフェースについて簡単に説明します。最も重要な相違点は次のものです。

- 外部デバッグインタフェースは、外部デバッガがどのように V7 デバッグリソースにアクセスするかを定義します。
- プロセッサインタフェースは、ARMv7 プロセッサがどのように自分のデバッグリソースにアクセスするかを記述します。

v7 デバックでは、*ARM デバッグインタフェース v5 アーキテクチャ仕様*(ADIv5) に基づいた外部デバッグインタフェースを推奨します。ADIv5 と、v6 デバッグおよび v6.1 デバッグで推奨されるインタフェースとの最も重要な相違点は、ADIv5 では電力オフの前後にわたるプロセッサのデバッグがサポートされていることです。

ARMv7 への準拠のために ADIv5 インタフェースは必須ではありませんが、ARM RealView® ツールを使用するには、このインタフェースが実装されている必要があります。

ADIv5 では、JTAG ワイヤインタフェースと、低ピン数の*シリアルワイヤ*(SW) インタフェースの両方がサポートされています。RealView ツールは、いずれのワイヤインタフェースもサポートしています。

ADIv5 インタフェースでは、ARMv7 プロセッサなどのデバッグオブジェクトが、一組のリソースを、メモリマップされたペリフェラルとして抽象化できます。デバッグリソースへのアクセスは、32 ビットの読み出し / 書き込み転送として行われます。電力オフ時のデバッグをサポートするため、特定のリソースが使用できないときには、そのリソースへのアクセスでエラー応答を返す抽象化が導入されています。これは、メモリマップされたペリフェラルが例外的な状況で、スレーブにより生成されたエラー応答を返すことができるのと同様の方式です。

v7 デバッグでは、プロセッサで実行されるソフトウェアが、すべてのデバッグレジスタにアクセス可能な必要があります。デバッグレジスタの特定の基本サブセットにアクセスを提供するため、v7 デバッグではベースラインコプロセッサ 14 (CP14) インタフェースの実装が必要です。*ベースライン CP14 デバッグレジスタ インタフェース*P. C6-32 を参照して下さい。他のデバッグレジスタへのアクセスを提供するため、v7 デバッグでは 2 つのオプションのいずれかが使用できます。

- 拡張 CP14 インタフェース。これは、v6 デバッグおよび v6.1 デバッグの要件とほぼ同じです。
- メモリマップされたインタフェース

実装に両方のオプションを含めることもできます。

ARMv7 では、デバッグ、トレース、パフォーマンスモニタのレジスタインタフェースの組み合わせすべてが許可されているわけではありません。ARMv7 実装の 3 つのオプションを、P. C1-10 の表 C1-2 に示します。多くの場合、オプションのメモリマップされたインタフェースが許可されます。これらは括弧で囲んで示されています。デバッグインタフェースまたはトレースインタフェースで、オプションのメモリマップされたインタフェースを実装する場合、これらのインタフェースの両方に実装することをお勧めします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C1-2 Options for interfacing to debug in ARMv7

Processor interface to debug registers	Processor interface to trace registers	Processor interface to performance monitor
Baseline CP14 + Memory-mapped	(Memory-mapped)[a]	CP15
Baseline CP14 + Extended CP14 (+ Memory-mapped)[a]	Memory-mapped)[a]	CP15
Baseline CP14 + Extended CP14 (+ Memory-mapped)[a]	CP14 (+ Memory-mapped)[a]	CP15

a. Interfaces shown in brackets are optional, see text for more information.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 C1-2 ARMv7 でのデバッグインタフェースのオプション

プロセッサからデバッグレジスタへのインタフェース	プロセッサからトレースレジスタへのインタフェース	プロセッサからパフォーマンスモニタへのインタフェース
ベースライン CP14 + メモリマップ	(メモリマップ)[a]	CP15
ベースライン CP14 + 拡張 CP14（+ メモリマップ)[a]	(メモリマップ)[a]	CP15
ベースライン CP14 + 拡張 CP14（+ メモリマップ)[a]	CP14（+ メモリマップ)[a]	CP15

a. 括弧で囲まれているインタフェースはオプションです。詳細については本文を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter C2
Invasive Debug Authentication

This chapter describes the authentication controls on invasive debug operations. It contains the following section:

- *About invasive debug authentication* on page C2-2.

Note

The recommended external debug interface provides an authentication interface that controls both invasive debug and non-invasive debug, as described in *Authentication signals* on page AppxA-3. This chapter describes how you can use this interface to control invasive debug. For information about using the interface to control non-invasive debug see Chapter C7 *Non-invasive Debug Authentication*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 C2 章
侵襲性デバッグの認証

本章では、侵襲性デバッグ操作に対する認証制御について説明します。本章は次のセクションから構成されています。

- *侵襲性デバッグの認証について*P. C2-2

注

推奨される外部デバッグインタフェースでは、*認証信号*P. 付録 A-3 に示されているように、侵襲性デバッグと非侵襲性デバッグの両方を制御する認証インタフェースが提供されます。本章では、このインタフェースを使用して侵襲性デバッグを制御する方法について説明します。このインタフェースを使用して非侵襲性デバッグを制御する方法については、第 C7 章*「非侵襲性デバッグの認証」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C2.1 About invasive debug authentication

Invasive debug can be enabled or disabled. If it is disabled the processor ignores all debug events except BKPT Instruction. This means that debug events other than the BKPT Instruction debug event do not cause the processor to enter Debug state or to take a debug exception.

In addition, if a processor implements the Security Extensions, invasive debug can be permitted or not permitted. When invasive debug is not permitted, all debug events are not permitted. When a debug event is not permitted:

- if the debug event is not a BKPT Instruction debug event then it is ignored
- if the debug event is a BKPT Instruction debug event then it causes a debug exception.

Note

The BKPT Instruction debug event is never ignored.

The difference between enabled and permitted is that whether a debug event is permitted depends on both the security state and the operating mode of the processor.

For debug events that cause entry to Debug state, *Secure User halting debug* refers to permitting these events in Secure User mode when invasive debug is not permitted in Secure privileged modes. The debug events that cause entry to Debug state are:

- Halting debug events
- if Halting debug-mode is selected, Software debug events.

Support for Secure User halting debug is required in v6.1 Debug. In v7 Debug it is IMPLEMENTATION DEFINED whether Secure User halting debug is supported. On an implementation that does not support Secure User halting debug the DBGDIDR.nSUHD_imp bit is RAO, see *Debug ID Register (DBGDIDR)* on page C10-3. ARM deprecates the use of Secure User halting debug.

If the Security Extensions are implemented, when invasive debug is not permitted in Secure privileged modes it must be possible to permit, in Secure User mode, the debug events that do not cause entry to Debug state. The debug events that do not cause entry to Debug state are Software debug events when Monitor debug-mode is selected.

Note

When the Security Extensions are implemented, the Debug architecture distinguishes between permitting invasive halting debug and permitting invasive non-halting debug. However, in Non-secure state and in Secure privileged modes whether a debug event is permitted does not depend on whether the event would cause entry to Debug state. Therefore, the distinction between permitting invasive halting debug and invasive non-halting debug applies only in Secure User mode.

When Secure User halting debug is supported, the processor can be configured so that both invasive halting debug and invasive non-halting debug are permitted in Secure User mode when invasive debug is not permitted in Secure privileged modes. Therefore, the alternatives for when a debug event is permitted are:

- in all processor modes, in both Secure and Non-secure security states
- only in Non-secure state

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C2.1 侵襲性デバッグの認証について

侵襲性デバッグは、可能または不可能にすることができます。不可能の場合、プロセッサは BKPT 命令を除くすべてのデバッグイベントを無視します。この場合、BKPT 命令以外のデバッグイベントでは、プロセッサがデバッグ状態に遷移したり、デバッグ例外を取得したりすることはありません。

さらに、プロセッサにセキュリティ拡張機能が実装されている場合、侵襲性デバッグを許可または禁止することができます。侵襲性デバッグが禁止されている場合、すべてのデバッグイベントは禁止されます。デバッグイベントが禁止されている場合、次の動作が適用されます。

- BKPT 命令以外のデバッグイベントは無視されます。
- BKPT 命令デバッグイベントはデバッグ例外を発生します。

注

BKPT 命令デバッグイベントが無視されることはありません。

可能と許可との相違点は、デバッグイベントの許可がセキュリティ状態とプロセッサの動作モードの両方に依存するかどうかです。

デバッグ状態への遷移を引き起こすデバッグイベントの場合、*セキュアユーザ ホールトデバッグ*は、侵襲性デバッグがセキュア特権モードで禁止されている場合でも、セキュア ユーザモードでこれらのイベントを許可することを意味します。デバッグ状態への遷移を引き起こすデバッグイベントには、次のものがあります。

- ホールトデバッグ イベント
- ホールト デバッグモードが選択されている場合、ソフトウェアデバッグ イベント

v6.1 デバッグでは、セキュアユーザ ホールトデバッグのサポートが必須です。v7 デバッグでは、セキュアユーザ ホールトデバッグをサポートするかどうかは実装定義です。セキュアユーザ ホールトデバッグをサポートしない実装では、DBGDIDR.nSUHD_imp ビットは RAO です。*デバッグ ID レジスタ (DBGDIDR)* P. C10-3 を参照して下さい。セキュアユーザ ホールトデバッグの使用は推奨されません。

セキュリティ拡張機能が実装されていて、セキュア特権モードで侵襲性デバッグが許可されていない場合、デバッグ状態への遷移を引き起こさないデバッグイベントを、セキュアユーザ モードで許可することが可能な必要があります。デバッグ状態への遷移を引き起こさないデバッグイベントは、モニタデバッグ モードが選択されている場合のソフトウェアデバッグ イベントです。

注

セキュリティ拡張機能が実装されている場合、デバッグアーキテクチャでは侵襲性ホールトデバッグの許可と、侵襲性非ホールトデバッグの許可とが区別されます。ただし、非セキュア状態およびセキュア特権モードでは、デバッグイベントが許可されるかどうかは、そのイベントがデバッグ状態への遷移を引き起こすかどうかには依存しません。このため、侵襲性ホールトデバッグの許可と、侵襲性非ホールトデバッグの許可との区別は、セキュア ユーザモードにのみ適用されます。

セキュアユーザ ホールトデバッグがサポートされている場合、セキュア ユーザモードでは侵襲性ホールトデバッグと侵襲性非ホールトデバッグの両方が許可され、セキュア特権モードでは侵襲性デバッグが許可されないように、プロセッサを構成可能です。このため、デバッグイベントが許可される条件は次のいずれかです。

- すべてのプロセッサモードで、セキュアおよび非セキュアセキュリティ状態の両方で
- 非セキュア状態のみで
- 非セキュア状態およびセキュアユーザ モードで

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- in Non-secure state and also in Secure User mode.

When Secure User halting debug is not supported, the processor can be configured only so that invasive non-halting debug is permitted in Secure User mode when invasive debug is not permitted in Secure privileged modes. Any debug event that would cause entry to Debug state is ignored, unless it is a BKPT Instruction debug event. Therefore, the alternatives for when a debug event is permitted are:

- in all processor modes, in both Secure and Non-secure security states
- only in Non-secure state
- in Non-secure state and also, if it will not cause entry to Debug state, in Secure User mode.

In v6.1 Debug and v7 Debug, invasive debug authentication can be controlled dynamically, meaning that whether a debug event is permitted can change while the processor is running, or while the processor is in Debug state. For more information, see *Generation of debug events* on page C3-40.

In v6 Debug, invasive debug authentication can be changed only while the processor is in reset.

In the recommended external debug interface, the signals that control the enabling and permitting of debug events are **DBGEN** and **SPIDEN**. **SPIDEN** is only implemented on processors that implement Security Extensions. See *Authentication signals* on page AppxA-3.

Part C of this manual assumes that the recommended external debug interface is implemented.

Note

- **DBGEN** and **SPIDEN** also control non-invasive debug, see *About non-invasive debug authentication* on page C7-2.
- For more information about use of the authentication signals see *Changing the authentication signals* on page AppxA-4.

If **DBGEN** is LOW, all invasive debug is disabled.

On processors that do not implement Security Extensions, if **DBGEN** is HIGH, invasive debug is enabled and permitted in all modes, see Table C2-1:

Table C2-1 Invasive debug authentication, Security Extensions not implemented

DBGEN	Modes in which invasive debug is permitted
LOW	None. Invasive debug is disabled.
HIGH	All modes.

On processors that implement the Security Extensions, if both **DBGEN** and **SPIDEN** are HIGH, invasive debug is enabled and all debug events are permitted in all modes and in both Secure and Non-secure security states.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

セキュアユーザ ホールトデバッグがサポートされていない場合、セキュアユーザ モードでは侵襲性非ホールトデバッグが許可され、セキュア特権モードでは侵襲性デバッグが許可されないように、プロセッサを構成可能です。デバッグ状態への遷移を引き起こすデバッグイベントは、BKPT 命令デバッグイベントを除いてすべて無視されます。このため、デバッグイベントが許可される条件は次のいずれかです。

- すべてのプロセッサモードで、セキュアおよび非セキュアセキュリティ状態の両方で
- 非セキュア状態のみで
- 非セキュア状態、および（デバッグ状態への遷移を引き起こさない場合に限り）セキュアユーザ モードで

v6.1 デバッグと v7 デバッグでは、侵襲性デバッグの認証を動的に制御可能です。つまり、デバッグイベントが許可されているかどうかをプロセッサの実行中や、プロセッサがデバッグ状態のときに変更できます。詳細については、*デバッグ イベントの生成* P. C3-40 を参照して下さい。

v6 デバッグでは、侵襲性デバッグの認証はプロセッサのリセット時にのみ変更可能です。

推奨される外部デバッグインタフェースでは、デバッグイベントの可能と許可を制御する信号は DBGEN と SPIDEN です。SPIDEN は、セキュリティ拡張機能を実装しているプロセッサにのみ実装されています。詳細については、*認証信号* P. 付録 A-3 を参照して下さい。

本書のパート C では、推奨される外部デバッグインタフェースの実装を想定しています。

注

- DBGEN および SPIDEN は、非侵襲性デバッグも制御します。*非侵襲性デバッグの認証について* P. C7-2 を参照して下さい。
- 認証信号の使用法の詳細については、*認証信号の変更* P. 付録 A-4 を参照して下さい。

DBGEN が LOW の場合、すべての侵襲性デバッグは不可能です。

セキュリティ拡張機能を実装していないプロセッサでは、DBGEN が HIGH の場合、侵襲性デバッグが可能で、すべてのモードで許可されています。表 C2-1 を参照して下さい。

表 C2-1 セキュリティ拡張機能が実装されていない場合の侵襲性デバッグの認証

DBGEN	侵襲性デバッグが許可されるモード
LOW	なし。侵襲性デバッグは不可能です。
HIGH	すべてのモード

セキュリティ拡張機能を実装しているプロセッサでは、DBGEN と SPIDEN の両方が HIGH の場合、侵襲性デバッグが可能で、すべてのモードにおいて、セキュアおよび非セキュアセキュリティ状態の両方ですべてのデバッグイベントが許可されます。

DBGEN が HIGH で SPIDEN が LOW の場合、次の状態を示します。

- 侵襲性デバッグが可能
- 非セキュア状態ですべてのデバッグイベントを許可

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If **DBGEN** is HIGH and **SPIDEN** is LOW:

- invasive debug is enabled
- all debug events are permitted in the Non-secure state
- no debug events are permitted in Secure privileged modes.
- whether invasive debug is permitted in Secure User mode depends on:
  - — the value of the SDER.SUIDEN bit, see *c1, Secure Debug Enable Register (SDER)* on page B3-108.
  - — if Secure User halting debug is not supported, whether the debug event would cause entry to Debug state.

This is shown in Table C2-2.

Table C2-2 Invasive debug authentication, Security Extensions implemented

DBGEN[a]	SPIDEN[a]	SUIDEN[b]	Mode	Security state	Invasive debug
LOW	X	X	Any	Either	Disabled
HIGH	LOW	0	Any	Non-secure	Enabled and permitted
				Secure	Enabled but not permitted
HIGH	LOW	1	Any	Non-secure	Enabled and permitted
			User	Secure	See note [c]
			Privileged	Secure	Enabled but not permitted
HIGH	HIGH	X	Any	Either	Enabled and permitted

a. Authentication signals, see *Authentication signals* on page AppxA-3.
b. SDER.SUIDEN bit, see *c1, Secure Debug Enable Register (SDER)* on page B3-108.
c. Invasive non-halting debug is permitted.
If Secure User halting debug is not supported then invasive halting debug is enabled but not permitted. Otherwise, invasive halting debug is enabled and permitted.

Note

Invasive and non-invasive debug authentication enable you to protect Secure processing from direct observation or invasion by an untrusted debugger. If you are designing a system you must be aware that security attacks can be aided by the invasive and non-invasive debug facilities. For example, Debug state or the DBGDSCR.INTdis register bit might be used for a denial of service attack, and the Non-secure performance monitors might be used to measure the side-effects of Secure processing on Non-secure code.

ARM recommends that where you are concerned about such attacks you disable invasive and non-invasive debug in all modes. However you must be aware of the limitations on the protection that debug authentication can provide, because similar attacks can be made by running malicious code on the processor in Non-secure state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- セキュア特権モードですべてのデバッグイベントを禁止
- セキュアユーザ モードで侵襲性デバッグが許可されるかどうかは、次の要素に依存します。
  - – SDER.SUIDEN ビットの値。*c1、セキュアデバッグ イネーブルレジスタ (SDER)* P. B3-108 を参照して下さい。
  - – セキュアユーザ ホールトデバッグがサポートされていない場合、デバッグイベントでデバッグ状態への遷移が発生するかどうか。

これを、表 C2-2 に示します。

表 C2-2 セキュリティ拡張機能が実装されている場合の侵襲性デバッグの認証

DBGEN[a]	SPIDEN[a]	SUIDEN[b]	モード	セキュリティ状態	侵襲性デバッグ
LOW	X	X	任意	両方	不可能
HIGH	LOW	0	任意	非セキュア	可能かつ許可
				セキュア	可能だが禁止
HIGH	LOW	1	任意	非セキュア	可能でかつ許可
			ユーザ	セキュア	注記[c] 参照
			特権	セキュア	可能だが禁止
HIGH	HIGH	X	任意	両方	可能でかつ許可

a. 認証信号。*認証信号* P. 付録 A-3 を参照して下さい。
b. SDER.SUIDEN ビット。*c1、セキュアデバッグ イネーブルレジスタ (SDER)* P. B3-108 を参照して下さい。
c. 侵襲性の非ホールトデバッグは許可されます。
セキュアユーザ ホールトデバッグがサポートされていない場合、侵襲性ホールトデバッグは可能ですが、禁止されます。それ以外の場合、侵襲性ホールトデバッグは可能でかつ許可されています。

注

侵襲性および非侵襲性デバッグの認証により、信用済みでないデバッガによりセキュアな処理が直接観測または侵襲されることを防止できます。システムを設計するときは、侵襲性および非侵襲性のデバッグ機構は、セキュリティへの攻撃に利用される恐れがあることに注意する必要があります。例えば、デバッグ状態や DBGDSCR.INTdis レジスタビットは DoS 攻撃に使用されることがあります。また、非セキュアなパフォーマンスモニタは、非セキュアなコードに対するセキュアな処理の副作用を測定するために使用されることがあります。

このような攻撃の恐れがある場所では、すべてのモードで侵襲性および非侵襲性デバッグを不可能にすることをお勧めします。ただし、非セキュア状態のプロセッサで悪意のコードを実行することでも同様な攻撃が可能なため、デバッグ認証による保護には限界があることも理解しておく必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter C3
Debug Events

This chapter describes debug events. Debug events trigger invasive debug operations. It contains the following sections:

- *About debug events* on page C3-2
- *Software debug events* on page C3-5
- *Halting debug events* on page C3-38
- *Generation of debug events* on page C3-40
- *Debug event prioritization* on page C3-43.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 C3 章
デバッグイベント

本章では、デバッグイベントについて説明します。デバッグイベントは、侵襲性デバッグ動作をトリガするものです。本章は次のセクションから構成されています。

- *デバッグイベントについて* P. C3-2
- *ソフトウェアデバッグ イベント* P. C3-5
- *ホールトデバッグ イベント* P. C3-38
- *デバッグ イベントの生成* P. C3-40
- *デバッグイベントの優先度付け* P. C3-43

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C3.1 About debug events

A debug event can be either:

- A Software debug event, see *Software debug events* on page C3-5
- A Halting debug event, see *Halting debug events* on page C3-38.

A processor responds to a debug event in one of the following ways:

- ignores the debug event
- takes a debug exception, see Chapter C4 *Debug Exceptions*
- enters Debug state, see Chapter C5 *Debug State*.

The response depends on the configuration. This is shown in Table C3-1 and in:

- Figure C3-1 on page C3-3 for v7 Debug
- Figure C3-2 on page C3-4 for v6 Debug and v6.1 Debug.

Table C3-1 Processor behavior on debug events

Configuration		Behavior, for specified debug event			
Enabled and permitted[a]	**DBGDSCR [15:14]**[b]	**BKPT Instruction debug event**	**Other Software debug event**	**Halting debug event**	**Debug-mode selected and enabled**
No	xx [c]	Debug exception [d]	Ignore	Ignore [e]	Disabled or not permitted
Yes	00	Debug exception [d]	Ignore	Debug state entry [f]	None
Yes	x1	Debug state entry	Debug state entry	Debug state entry	Halting
Yes	10	Debug exception	Debug exception or UNPREDICTABLE [g]	Debug state entry [f]	Monitor

a. Invasive debug is enabled and the debug event is permitted. Whether a debug event is permitted might depend on the type of debug event as well as the configuration of the processor, see Chapter C2 *Invasive Debug Authentication*.
b. See *Debug Status and Control Register (DBGDSCR)* on page C10-10.
c. The value of DBGSCR[15:14] is ignored when invasive debug is disabled or the debug event is not permitted. If debug is disabled these bits are RAZ.
d. When debug is disabled or the debug event is not permitted, the BKPT instruction generates a debug exception rather than being ignored. The DBGDSCR, IFSR and IFAR are set as if a BKPT Instruction debug exception occurred. See *Effects of debug exceptions on CP15 registers and the DBGWFAR* on page C4-4.
e. The processor might enter Debug state later, see *Halting debug events* on page C3-38.
f. In v6 Debug, it is IMPLEMENTATION DEFINED whether the processor enters Debug state or ignores the event.
g. Be careful when programming debug events when Monitor debug-mode is selected and enabled, because certain conditions can lead to UNPREDICTABLE behavior, see *Unpredictable behavior on Software debug events* on page C3-24. In v6 Debug and v6.1 Debug, some events are ignored in this state.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C3.1 デバッグイベントについて

デバッグイベントは、次のいずれかです。

- ソフトウェアデバッグ イベント。*ソフトウェアデバッグ イベント* P. C3-5 を参照して下さい。
- ホールトデバッグ イベント。*ホールトデバッグ イベント* P. C3-38 を参照して下さい。

プロセッサは、次のいずれかの方法でデバッグイベントに応答します。

- デバッグイベントを無視する。
- デバッグ例外を取得する。第 C4 章*「デバッグ例外」*を参照して下さい。
- デバッグ状態に遷移する。第 C5 章*「デバッグ状態」*を参照して下さい。

応答は構成によって異なります。構成と動作の関係を、表 C3-1 および次の図に示します。

- P. C3-3 の図 C3-1（v7 デバッグ）
- P. C3-4 の図 C3-2（v6 デバッグおよび v6.1 デバッグ）

表 C3-1 デバッグイベント時のプロセッサの動作

構成		指定のデバッグイベントに対する動作			
可能かつ許可[a]	**DBGDSCR [15:14]**[b]	**BKPT 命令デバッグイベント**	**他のソフトウェアデバッグ イベント**	**ホールトデバッグイベント**	**選択されており可能なデバッグモード**
いいえ	xx [c]	デバッグ例外 [d]	無視	無視 [e]	不可能または禁止
はい	00	デバッグ例外 [d]	無視	デバッグ状態遷移 [f]	なし
はい	x1	デバッグ状態遷移	デバッグ状態遷移	デバッグ状態遷移	遷移ホールト
はい	10	デバッグ例外	デバッグ例外または予測不能 [g]	デバッグ状態遷移 [f]	モニタ

a. 侵襲性デバッグは可能で、デバッグ イベントは許可されます。デバッグ イベントが許可されるかどうかは、デバッグ イベントのタイプと、プロセッサの構成の両方に依存します。第 C2 章*「侵襲性デバッグの認証」*を参照して下さい。
b. 詳細については、*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。
c. 侵襲性デバッグが不可能なとき、またはデバッグ イベントが禁止されているときは、DBGSCR[15:14] の値は無視されます。デバッグが不可能な場合、これらのビットは RAZ です。
d. デバッグが不可能またはデバッグ イベントが禁止されている場合、BKPT 命令は無視されず、デバッグ例外を生成します。DBGDSCR、IFSR、IFAR は、BKPT 命令によるデバッグ例外が発生した場合と同様に設定されます。詳細については、*CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響* P. C4-4 を参照して下さい。
e. プロセッサは、後でデバッグ状態に遷移する可能性があります。*ホールトデバッグ イベント* P. C3-38 を参照して下さい。
f. v6 デバッグでは、プロセッサがデバッグ状態に遷移するか、イベントを無視するかは実装定義です。
g. モニタデバッグ モードが選択され可能なとき、特定の条件は予測不能な動作を引き起こす可能性があるため、デバッグ イベントのプログラミングに注意して下さい。*ソフトウェアデバッグ イベントでの予測不能な動作* P. C3-24 を参照して下さい。v6 デバッグおよび v6.1 デバッグでは、この状態で一部のイベントが無視されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure C3-1 Processor behavior on debug events, for v7 Debug

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

図 C3-1 デバッグイベント時のプロセッサの動作、v7 デバッグ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure C3-2 Processor behavior on debug events, for v6 Debug and v6.1 Debug

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

図 C3-2 デバッグイベント時のプロセッサの動作、v6 デバッグおよび v6.1 デバッグ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C3.2 Software debug events

A Software debug event can be any of the following:

- A Breakpoint debug event, see *Breakpoint debug events*
- A Watchpoint debug event, see *Watchpoint debug events* on page C3-15
- A BKPT Instruction debug event, see *BKPT Instruction debug events* on page C3-20
- A Vector Catch debug event, see *Vector Catch debug events* on page C3-20.

Memory addresses on page C3-23 describes the addresses used for generating Software debug events in different memory system implementations.

If Monitor debug-mode is selected and enabled, the behavior of certain types of Software debug event is UNPREDICTABLE. For more information, see *Unpredictable behavior on Software debug events* on page C3-24.

Pseudocode details of Software debug events on page C3-27 gives pseudocode for the operation of the Software debug events.

C3.2.1 Breakpoint debug events

A Breakpoint debug event is defined by a pair of registers described to as a *Breakpoint Register Pair* (BRP), comprising a *Breakpoint Control Register* (DBGBCR) and a *Breakpoint Value Register* (DBGBVR). BRPs, DBGBCRs, and DBGBVRs number upwards from 0, with BRPn comprising DBGBCRn and DBGBVRn. For details of the breakpoint registers see:

- *Breakpoint Control Registers (DBGBCR)* on page C10-49
- *Breakpoint Value Registers (DBGBVR)* on page C10-48.

The DBGDIDR.BRPs field specifies the number of BRPs implemented, see *Debug ID Register (DBGDIDR)* on page C10-3. The maximum number of BRPs is 16.

You can define a Breakpoint debug event:

- Based on comparison of an *Instruction Virtual Address* (IVA) with the value held in a DBGBVR. See *Memory addresses* on page C3-23 for the definition of an IVA.
- Based on comparison of the Context ID with the value held in a DBGBVR. Some BRPs might not support Context ID comparison. The DBGDIDR.CTX_CMPs field specifies the number of BRPs that support Context ID comparison, see *Debug ID Register (DBGDIDR)* on page C10-3.
- By linking a BRP to a second BRP, to define a single Breakpoint debug event. One pair includes an IVA for comparison, and the second pair includes a Context ID value.

In all cases, the DBGBCR defines some additional conditions that must be met for the BRP to generate a Breakpoint debug event, including whether the BRP is enabled.

The terms *hit* and *miss* are used to describe whether the conditions defined in the BRP are met:

- a hit occurs when the conditions are met
- a miss occurs when a condition is not met, meaning the processor does not generate a debug event.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C3.2 ソフトウェアデバッグ イベント

ソフトウェアデバッグ イベントは次のいずれかです。

- ブレークポイント デバッグイベント。*ブレークポイントデバッグ イベント*を参照して下さい。
- ウォッチポイント デバッグイベント。*ウォッチポイントデバッグ イベント* P. C3-15 を参照して下さい。
- BKPT 命令デバッグイベント。*BKPT 命令デバッグイベント* P. C3-20 を参照して下さい。
- ベクタキャッチ デバッグイベント。*ベクタキャッチデバッグ イベント* P. C3-20 を参照して下さい。

各種のメモリシステムの実装で、ソフトウェアデバッグイベントを生成するために使用されるアドレスについては、*メモリアドレス* P. C3-23 を参照して下さい。

モニタデバッグ モードが選択され可能な場合、特定のタイプのソフトウェアデバッグ イベントの動作は予測不能です。詳細については、*ソフトウェアデバッグ イベントでの予測不能な動作* P. C3-24 を参照して下さい。

ソフトウェアデバッグ イベント操作用の擬似コードについては、*ソフトウェアデバッグ イベントの擬似コードの詳細* P. C3-27 を参照して下さい。

C3.2.1 ブレークポイントデバッグ イベント

ブレークポイントデバッグ イベントは、*ブレークポイント制御レジスタ*(DBGBCR) と*ブレークポイント値レジスタ*(DBGBVR) によって定義されます。これらのレジスタは総称として*ブレークポイント レジスタ ペア*(BRP) と呼ばれます。BRP、DBGBCR、DBGBVR の数は 0 以上で、BRPn は DBGBCRn と DBGBVRn で構成されます。ブレークポイントレジスタの詳細については、次のセクションを参照して下さい。

- *ブレークポイント制御レジスタ (DBGBCR)* P. C10-49
- *ブレークポイント値レジスタ (DBGBVR)* P. C10-48

DBGDIDR.BRP のフィールドは、実装されている BRP の数を指定します。*デバッグ ID レジスタ (DBGDIDR)* P. C10-3 を参照して下さい。BRP の最大数は 16 です。

ブレークポイントデバッグ イベントは、次の方法で定義できます。

- *命令仮想アドレス*(IVA) と、DBGBVR に保持されている値との比較に基づいて。IVA の定義については、*メモリアドレス* P. C3-23 を参照して下さい。
- コンテキスト ID と、DBGBVR に保持されている値との比較に基づいて。一部の BRP では、コンテキスト ID の比較がサポートされていません。DBGDIDR.CTX_CMP のフィールドは、コンテキスト ID の比較をサポートする BRP の数を指定します。*デバッグ ID レジスタ (DBGDIDR)* P. C10-3 を参照して下さい。
- BRP を 2 番目の BRP にリンクし、単一のブレークポイントデバッグ イベントを定義して。1 つのペアには比較用の IVA が、2 番目のペアにはコンテキスト ID の値が含まれます。

いずれの場合でも、DBGBCR にはいくつかの追加条件が定義され、BRP がブレークポイントデバッグ イベントを生成するにはそれらの条件が満たされる必要があります。BRP が稼働中であるかどうかもこの条件に含まれます。

*ヒット*および*ミス*という用語は、BRP で定義されている条件が満たされたかどうかを示すために使用されます。

- 条件が満たされた場合、ヒットが発生します。
- 条件が満たされていない場合、ミスが発生します。この場合、プロセッサはデバッグイベントを生成しません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The following sections describe Breakpoint debug events:

- *Generation of Breakpoint debug events*
- *Debug event generation conditions defined by the DBGBCR* on page C3-7
- *IVA comparisons for Debug event generation* on page C3-8
- *IVA comparisons and instruction length* on page C3-10
- *Context ID comparisons for Debug event generation* on page C3-13
- *Additional considerations for IVA mismatch breakpoints* on page C3-13
- *Additional conditions for linked BRPs* on page C3-15.

Generation of Breakpoint debug events

For each instruction in the program flow, the debug logic tests all the BRPs. For each BRP, the debug logic generates a Breakpoint debug event only if all of the following apply:

- When the BRP is tested, the conditions specified in the DBGBCR are met, see *Debug event generation conditions defined by the DBGBCR* on page C3-7.
- The comparison with the value in the DBGBVR is successful. When two BRPs are linked to define a single Breakpoint debug event, both comparisons must succeed. For more information see:
  - *IVA comparisons for Debug event generation* on page C3-8
  - *Context ID comparisons for Debug event generation* on page C3-13.
- The instruction is committed for execution.

  Note

  The processor must test for any possible Breakpoint debug events before it executes the instruction. The debug logic might test the BRPs when an instruction is prefetched. However, it must not generate a Breakpoint debug event if the instruction is not committed for execution.

If all of these conditions are met, the debug logic generates the Breakpoint debug event regardless of whether the instruction passes its condition code test.

In ARMv6 and the ARMv7-A and ARMv7-R architecture profiles, the debug logic generates the debug event regardless of the type of instruction.

Breakpoint debug events are synchronous. That is, the debug event acts like an exception that cancels the breakpointed instruction.

When invasive debug is enabled and Monitor debug-mode is selected, if Breakpoint debug events are permitted a Breakpoint debug event generates a Prefetch Abort exception.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ブレークポイントデバッグ イベントの説明については、次のセクションを参照して下さい。

- *ブレークポイントデバッグ イベントの生成*
- *DBGBCR により定義されるデバッグイベント生成条件* P. C3-7
- *デバックイベント生成のための IVA 比較* P. C3-8
- *IVA 比較と命令長* P. C3-10
- *デバックイベント生成のためのコンテキスト ID 比較* P. C3-13
- *IVA 不一致ブレークポイントについての追加の考慮事項* P. C3-13
- *リンク付き BRP についての追加条件* P. C3-15

ブレークポイントデバッグ イベントの生成

プログラムフローの各命令について、デバッグロジックはすべての BRP をテストします。各 BRP について、デバッグロジックは次のすべての条件が適用される場合のみブレークポイントデバッグ イベントを生成します。

- BRP をテストしたとき、DBGBCR に指定されている条件が満たされている。*DBGBCR により定義されるデバッグイベント生成条件* P. C3-7 を参照して下さい。
- DBGBVR の値との比較が成功した。2 つの BRP がリンクされて単一のブレークポイントデバッグ イベントを定義している場合、両方の比較が成功する必要があります。詳細については、次のセクションを参照して下さい。
  - *デバックイベント生成のための IVA 比較* P. C3-8
  - *デバックイベント生成のためのコンテキスト ID 比較* P. C3-13
- 命令が実行にコミットされている。

  注

  プロセッサは、命令を実行する前に、可能なブレークポイントデバッグ イベントのすべてをテストする必要があります。デバッグロジックは、命令のプリフェッチ時に BRP をテストする可能性があります。ただし、命令が実行にコミットされていない場合にブレークポイントデバッグ イベントを生成することはできません。

これらの条件すべてが満たされた場合、命令が条件コードテストにパスしたかどうかにかかわらず、デバッグロジックはブレークポイントデバッグ イベントを生成します。

ARMv6、ARMv7-A、ARMv7-R アーキテクチャプロファイルでは、デバッグロジックは命令のタイプにかかわらずデバッグイベントを生成します。

ブレークポイントデバッグ イベントは同期です。つまり、デバッグイベントは、ブレークポイントが設定された命令をキャンセルする例外と同じように動作します。

侵襲性デバッグが可能で、モニタデバッグ モードが選択されている場合、ブレークポイントデバッグ イベントが許可されていれば、ブレークポイントデバッグ イベントはプリフェッチアボート例外を生成します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Debug event generation conditions defined by the DBGBCR

For each BRP, the DBGBCR defines some conditions for generating a Breakpoint debug event, using the following register fields:

Breakpoint enable

Controls whether this BRP is enabled.

Privileged mode control

Controls whether this BRP defines a Breakpoint debug event that can occur:

- only in User mode
- only in a privileged mode
- only in User, System or Supervisor modes
- in any mode.

Security state control

If the processor implements the Security Extensions, this field controls whether this BRP defines a Breakpoint debug event that can occur only in Secure state, only in Non-secure state, or in either security state.

For more information, including the differences in different versions of the Debug architecture, see *Breakpoint Control Registers (DBGBCR)* on page C10-49.

When two BRPs are linked to define a single Breakpoint debug event, the BRP that defines the IVA comparison also defines the privileged mode control and security state control, see *Additional conditions for linked BRPs* on page C3-15 for more information.

Other information in the DBGBCR

In addition to defining these conditions for generating a Breakpoint debug event, the DBGBCR controls the following:

- The DBGBVR meaning field defines the breakpoint type. The following sections describe all of the breakpoint types:
  - *IVA comparisons for Debug event generation* on page C3-8
  - *Context ID comparisons for Debug event generation* on page C3-13.
- The Linked BRP number field specifies whether the BRP is linked to another BRP. If this BRP is linked, this field gives the number of the linked BRP. For more information see *Additional conditions for linked BRPs* on page C3-15.
- For an IVA comparison, the DBGBVR defines a word-aligned address, and the Byte address select field specifies the bytes in that word that comprise the breakpointed instruction, see *IVA comparisons for Debug event generation* on page C3-8.
- For an IVA comparison in v7 Debug, the Address range mask field optionally specifies a bitmask that defines the low-order bits of the IVA and DBGBVR values that are excluded from the comparison, see *IVA comparisons for Debug event generation* on page C3-8.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DBGBCR により定義されるデバッグイベント生成条件

各 BRP について、DBGBCR は次のレジスタフィールドを使用して、ブレークポイントデバッグイベントを生成するためのいくつかの条件を定義します。

ブレークポイントイネーブル

この BRP が可能かどうかを制御します。

特権モード制御

この BRP が定義しているブレークポイントが、どのモードで発生可能かを制御します。

- ユーザモードのみ
- 特権モードのみ
- ユーザ、システム、スーパバイザモードのみ
- すべてのモード

セキュリティ状態制御

プロセッサにセキュリティ拡張機能が実装されている場合、このフィールドは、この BRP で定義しているブレークポイントデバッグ イベントがセキュア状態のみ、非セキュア状態のみ、または両方のセキュリティ状態のいずれで発生可能かを定義します。

デバッグアーキテクチャのバージョン間の相違点を含む詳細については、*ブレークポイント制御レジスタ (DBGBCR)* P. C10-49 を参照して下さい。

2 つの BRP がリンクされて単一のブレークポイントデバッグ イベントを定義する場合、IVA 比較を定義する BRP は、特権モード制御とセキュリティ状態制御も定義します。詳細については、*リンク付き BRP についての追加条件* P. C3-15 を参照して下さい。

DBGBCR に含まれる他の情報

ブレークポイントデバッグ イベントを生成するための上述の条件を定義するほか、DBGBCR は次の条件も制御します。

- DBGBVR 意味フィールドは、ブレークポイントのタイプを定義します。次のセクションでは、ブレークポイントのすべてのタイプについて説明されています。
  - *デバックイベント生成のための IVA 比較* P. C3-8
  - *デバックイベント生成のためのコンテキスト ID 比較* P. C3-13
- リンク付き BRP 番号フィールドは、BRP が他の BRP にリンクされているかどうかを指定します。この BRP がリンクされている場合、このフィールドにはリンク先の BRP の番号が指定されます。詳細については、*リンク付き BRP についての追加条件* P. C3-15 を参照して下さい。
- IVA 比較について、DBGBVR はワードアラインドのアドレスを定義し、バイトアドレス選択フィールドは、そのワードのうちどのバイトがブレークポイント対象の命令を構成するかを指定します。*デバックイベント生成のための IVA 比較* P. C3-8 を参照して下さい。
- v7 デバッグの IVA 比較について、アドレス範囲マスクフィールドは、比較から除外される IVA および DBGBVR 値の下位ビットを定義します（オプション）。*デバックイベント生成のための IVA 比較* P. C3-8 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

For IVA comparison in v7 Debug, you must use either byte address selection or address range masking to restrict the comparison made. However, you cannot use both at the same time.

IVA comparisons for Debug event generation

The result of an IVA comparison depends on the value in the DBGBVR either matching or mismatching the IVA value. In each case, you can link the BRP to a second BRP that defines a Context ID comparison. This means that the breakpoint types that depend on an IVA comparison are:

- Unlinked IVA match
- Unlinked IVA mismatch
- Linked IVA match
- Linked IVA mismatch.

When the DBGBCR is programmed for one of these breakpoint types, the debug logic generates a Breakpoint debug event only if all the other conditions for the breakpoint are met, and the IVA comparison is successful. That is, all other conditions are met and, taking account of any masking:

- for an IVA match, the IVA value equals the value in the DBGBVR
- for an IVA mismatch, the IVA value does not equal the value in the DBGBVR.

In the linked cases, the debug logic generates a Breakpoint debug event only if all the other conditions for the breakpoint are met, the IVA comparison is successful, and the Context ID comparison in the linked BRP is successful, see *Context ID comparisons for Debug event generation* on page C3-13. See *Additional conditions for linked BRPs* on page C3-15 for more information.

All versions of the Debug architecture support byte address selection, to specify which bytes of the word addressed by DBGBVR comprise the breakpointed instruction. v7 Debug supports an alternative bit masking scheme referred to as address range masking. The following subsections give more information about the IVA comparisons:

- *Condition for breakpoint generation on IVA match, without address range masking* on page C3-9
- *Condition for breakpoint generation on IVA mismatch, without address range masking* on page C3-9
- *Breakpoint address range masking behavior, v7 Debug* on page C3-9.

DBGBVR values must be word-aligned, and DBGBVR[1:0] are never used for IVA comparison. ARM instructions are always word-aligned, and therefore a DBGBVR value can specify exactly the IVA of an ARM instruction. See *IVA comparisons and instruction length* on page C3-10 for more information about how the instruction length affects how you must define a breakpoint.

Note

- v6 Debug does not support IVA mismatch.
- If it is supported, you can use IVA mismatch to generate a Breakpoint debug event when the processor executes an instruction other than the instruction indicated by the DBGBVR. You can use this for single-stepping, or for breakpointing all instructions outside a range of instruction addresses.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

v7 デバッグでの IVA 比較では、行われる比較を制限するためにバイトアドレス選択またはアドレス範囲マスクを使用する必要があります。ただし、両方を同時に使用することはできません。

デバックイベント生成のための IVA 比較

IVA 比較の結果は、DBGBVR の値が IVA の値と一致するか、しないかに依存します。いずれの場合も、BRP を、コンテキスト ID 比較を定義する 2 番目の BRP にリンクできます。つまり、IVA 比較に依存するブレークポイントのタイプは次のいずれかです。

- リンクなしの IVA 一致
- リンクなしの IVA 不一致
- リンク付きの IVA 一致
- リンク付きの IVA 不一致

これらのブレークポイントタイプのうち 1 つについて DBGBCR がプログラムされている場合、ブレークポイントの他の条件がすべて満たされ、IVA 比較が成功した場合のみ、デバッグロジックがブレークポイントデバッグ イベントを生成します。つまり、他の条件すべてが満たされ、かつマスクを考慮して次の条件が満たされる必要があります。

- IVA 一致の場合、IVA の値が DBGBVR と等しいこと
- IVA 不一致の場合、IVA の値が DBGBVR と等しくないこと

リンク付きの場合は、ブレークポイントに関する他の条件がすべて満たされ、IVA 比較が成功し、リンク付き BRP でコンテキスト ID 比較が成功した場合のみ、デバッグロジックがブレークポイントデバッグ イベントを生成します。*デバックイベント生成のためのコンテキスト ID 比較* P. C3- 13 を参照して下さい。詳細については、*リンク付き BRP についての追加条件* P. C3- 15 を参照して下さい。

デバッグアーキテクチャのすべてのバージョンはバイトアドレス選択をサポートしているため、DBGBVR によってアドレス指定されているワードのうち、どのバイトがブレークポイント対象の命令を構成するかを指定できます。v7 デバックでは、アドレス範囲マスクと呼ばれる別のビットマスク方式がサポートされています。次のサブセクションでは、IVA 比較についてより詳細に説明しています。

- *アドレス範囲マスクがない場合に IVA 一致でブレークポイントを生成するための条件* P. C3- 9
- *アドレス範囲マスクがない場合に IVA 不一致でブレークポイントを生成するための条件* P. C3- 9
- *v7 デバッグでのブレークポイントアドレス範囲マスクの動作* P. C3- 9

DBGBVR の値はワードアラインドの必要があり、DBGBVR[1:0] は IVA 比較に使用されることはありません。ARM 命令は必ずワードアラインドなので、DBGBVR の値は ARM 命令の IVA を正確に指定できます。命令の長さがブレークポイント定義の方法に与える影響の詳細については、*IVA 比較と命令長* P. C3- 10 を参照して下さい。

注

- v6 デバッグでは、IVA 不一致はサポートされていません。
- IVA 不一致がサポートされている場合、この機能を使用して、DBGBVR によって示される以外の命令をプロセッサが実行しているときに、ブレークポイントデバッグ イベントを生成することができます。この方法は、シングルステップ実行や、一定範囲のアドレス外に存在するすべての命令にブレークポイントを設定するために使用できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Condition for breakpoint generation on IVA match, without address range masking

When BRPn is programmed for IVA match, without address range masking, and all other conditions for generating a breakpoint are met, a Breakpoint debug event is generated only if both:

- bits [31:2] of the IVA are equal to the value of bits [31:2] of DBGBVRn
- the Byte address select field, bits [8:5], of DBGBCRn is programmed for an IVA match for the current Instruction set state and IVA[1:0] value, see *Byte address selection behavior on IVA match or mismatch* on page C10-55.

Note

In v7 Debug, to perform IVA comparison without address range masking you must set DBGBCR[28:24], the Address range mask field, to zero.

Condition for breakpoint generation on IVA mismatch, without address range masking

When BRPn is programmed for IVA mismatch, without address range masking, and all other conditions for generating a breakpoint are met, a Breakpoint debug event is generated only if either:

- bits [31:2] of the IVA are not equal to the value of bits [31:2] of DBGBVRn
- the Byte address select field, bits [8:5], of DBGBCRn is programmed for an IVA mismatch for the current Instruction set state and IVA[1:0] value, see *Byte address selection behavior on IVA match or mismatch* on page C10-55.

Note

In v7 Debug, to perform IVA comparison without address range masking you must set DBGBCR[28:24], the Address range mask field, to zero.

Breakpoint address range masking behavior, v7 Debug

When BRPn is programmed for IVA matching, the comparison is masked using the value held in the Address range mask field, DBGBCRn[28:24].

You can use the Address range mask field when programming the BRP for IVA mismatch, that is, when DBGBCR[28:24] != 0b00000 and DBGBCR[22] == 1. In this case, the address comparison portion of breakpoint generation hits for all addresses outside the masked address region.

If an implementation does not support breakpoint address range masking, the Address range mask field is RAZ.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

アドレス範囲マスクがない場合に IVA 一致でブレークポイントを生成するための条件

BRPn がアドレス範囲マスクなしの IVA 一致用にプログラムされており、ブレークポイント生成の他のすべての条件が満たされているとき、次の条件の両方が満たされている場合のみブレークポイントデバッグ イベントが生成されます。

- IVA のビット [31:2] が、DBGBVRn のビット [31:2] の値と等しい。
- DBGBCRn のバイトアドレス選択フィールドであるビット [8:5] が、現在の命令セット状態および IVA[1:0] の値について IVA 一致用にプログラムされている。*IVA 一致または不一致時のバイトアドレス選択の動作* P. C10-55 を参照して下さい。

注

v7 デバッグでは、アドレス範囲マスクなしで IVA 比較を実行するには、アドレス範囲マスクフィールドである DBGBCR[28:24] を 0 に設定する必要があります。

アドレス範囲マスクがない場合に IVA 不一致でブレークポイントを生成するための条件

BRPn がアドレス範囲マスクなしの IVA 不一致用にプログラムされており、ブレークポイント生成の他のすべての条件が満たされているとき、次の条件のいずれかが満たされている場合のみブレークポイントデバッグ イベントが生成されます。

- IVA のビット [31:2] が、DBGBVRn のビット [31:2] の値と等しくない。
- DBGBCRn のバイトアドレス選択フィールドであるビット [8:5] が、現在の命令セット状態および IVA[1:0] の値について IVA 不一致用にプログラムされている。*IVA 一致または不一致時のバイトアドレス選択の動作* P. C10-55 を参照して下さい。

注

v7 デバッグでは、アドレス範囲マスクなしで IVA 比較を実行するには、アドレス範囲マスクフィールドである DBGBCR[28:24] を 0 に設定する必要があります。

v7 デバッグでのブレークポイントアドレス範囲マスクの動作

BRPn が IVA 一致用にプログラムされている場合、アドレス範囲マスクフィールドである DBGBCRn[28:24] に保持されている値を使用して、比較がマスクされます。

アドレス範囲マスクフィールドは、BRP を IVA 不一致用にプログラムするとき、つまり DBGBCR[28:24] != 0b00000 かつ DBGBCR[22] == 1 の場合に使用できます。この場合、ブレークポイント生成のアドレス比較の部分は、マスクされたアドレス領域外のすべてのアドレスにヒットします。

実装でブレークポイントのアドレス範囲マスクがサポートされていない場合、アドレス範囲マスクフィールドは RAZ です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

There is no encoding for a full 32-bit mask. This mask would have the effect of setting a breakpoint that hits on every address comparison, and you can achieve this by setting:

- DBGBCR[22] to 1 to select an IVA mismatch
- DBGBCR[8:5] to 0b0000.

To use address range masking, you must also set DBGBCR[8:5], the Byte address field, to 0b1111.

IVA comparisons and instruction length

An instruction set is *fixed-length* if all of its instructions have the same length, and *variable-length* otherwise. In a variable-length instruction set a single instruction comprises one or more units of memory.

The ARM instruction set is an example of a fixed-length instruction set. In the ARM instruction set the size of each instruction is one word, and ARM instructions are always word aligned.

The following are examples of variable-length instruction sets:

- The ThumbEE instruction set, and the Thumb instruction set from ARMv6T2 onwards. In these instruction sets an instruction comprises one or two halfwords.
- Java bytecodes. A single Java bytecode comprises one or more bytes.

Before ARMv6T2, an implementation can treat the Thumb instruction set as a fixed-length 16-bit instruction set, as described in *BL and BLX (immediate) instructions, before ARMv6T2* on page AppxG-4. An implementation that does this can permit an exception to be taken between the two halfwords of a `BL` or `BLX` (immediate) instruction.

In a variable-length instruction set, for an instruction consisting of more than one unit of memory, the first unit of the instruction is defined as the unit of the instruction with the lowest address in memory.

In a fixed-length instruction set, an instruction consists of a single unit of memory. This unit is also the first unit of the instruction.

Instruction length considerations depend on the Debug architecture version, as described in the following subsections:

- *Effect of instruction length in v7 Debug*
- *Effect of instruction length in v6 Debug and v6.1 Debug* on page C3-11.

IVA comparison programming examples on page C3-12 gives examples of Breakpoint programming, taking account of possible instruction length effects, for all versions of the Debug architecture.

Effect of instruction length in v7 Debug

In v7 Debug there are four types of IVA breakpoint:

- IVA match with no address range mask, described as a *regular* IVA breakpoint
- IVA mismatch with no address range mask, described as a *step-off* IVA breakpoint
- IVA match with an address range mask, described as an *included range* IVA breakpoint

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

完全な 32 ビットマスクのエンコードは存在しません。このマスクは、すべてのアドレス比較にヒットするブレークポイントを設定するのと同じ効果を持ち、次の設定により実現できます。

- DBGBCR[22] を 1 にセットして IVA 不一致を選択する。
- DBGBCR[8:5] を 0b0000 に設定する。

アドレス範囲マスクを使用するには、バイトアドレス フィールドの DBGBCR[8:5] も 0b1111 に設定する必要があります。

IVA 比較と命令長

命令セットのすべての命令が同じ長さのとき、その命令セットを*固定長*と呼び、それ以外の場合は*可変長*と呼びます。可変長命令セットでは、単一の命令が 1 つまたは複数のメモリ単位で構成されます。

ARM 命令セットは、固定長命令セットの例です。ARM 命令セットでは各命令のサイズは 1 ワードで、ARM 命令は必ずワードアラインドです。

次に示すのは、可変長命令セットの例です。

- ThumbEE 命令セットと、ARMv6T2 およびそれ以降の Thumb 命令セット。これらの命令セットでは、命令が 1 つまたは 2 つのハーフワードで構成されます。
- Java バイトコード。1 つの Java バイトコードは、1 つまたは複数のバイトで構成されます。

ARMv6T2 以前の実装では、*ARM v 6T2 以前の BL および BLX（イミディエート）命令* P. 付録 G-4 に説明されているように、Thumb 命令セットを 16 ビットの固定長命令セットとして扱うことが可能です。この方式を使用する実装では、`BL` または `BLX`（イミディエート）命令の 2 つのハーフワードの中間で例外の取得を許可することができます。

可変長命令セットでは、メモリの複数の単位で構成される命令について、命令の最初の単位は、メモリの最も小さいアドレスに存在する命令単位と定義されます。

固定長命令セットでは、命令はメモリの 1 つの単位で構成されます。この単位は、命令の最初の単位でもあります。

命令長に関する考慮事項は、デバッグアーキテクチャのバージョンに依存します。詳細については、次に示すサブセクションを参照して下さい。

- *v7 デバッグでの命令長の影響*
- *v6 デバッグと v6.1 デバッグでの命令長の影響* P. C3-11

IVA 比較のプログラミング例 P. C3-12 には、デバッグアーキテクチャのすべてのバージョンについて、命令長が影響する可能性も考慮に入れたブレークポイントプログラミングの例が示されています。

v7 デバッグでの命令長の影響

v7 デバッグには、4 つのタイプの IVA ブレークポイントが存在します。

- アドレス範囲マスクなしの IVA 一致：*通常の* IVA ブレークポイントと呼びます。
- アドレス範囲マスクなしの IVA 不一致：*ステップオフ* IVA ブレークポイントと呼びます。
- アドレス範囲マスク付きの IVA 一致：*範囲付き* IVA ブレークポイントと呼びます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- IVA mismatch with an address range mask, described as an *excluded range* IVA breakpoint

Note

Support for address range masks on breakpoints is IMPLEMENTATION DEFINED.

For all types of IVA breakpoint, if the conditions in the DBGBCR are met, and the instruction is committed for execution, the BRP generates a Breakpoint debug event if the required DBGBVR comparison, taking account of the byte address selection and any address range masking, hits for the first unit of the instruction.

Table C3-2 shows the conditions for Breakpoint debug event generation by an instruction that comprises more than one unit of memory, assuming that the conditions in the DBGBCR are met and that the instruction is committed for execution.

Table C3-2 Breakpoint debug event generation for instructions of more than one unit of memory

DBGBVR comparison result[a]: First unit[b]	Any subsequent unit[b]	IVA breakpoint type	Breakpoint debug event generated?
Hit	-	Any	Yes
Miss	Hit	Regular, included range, or excluded range	UNPREDICTABLE
		Step-off	No
Miss	Miss	Any	No

a. Taking account of the byte address selection and any address range masking.
b. Of the instruction whose IVA is being compared.

Effect of instruction length in v6 Debug and v6.1 Debug

If the conditions in the DBGBCR are met, and the instruction is committed for execution, the BRP generates a Breakpoint debug event if the required DBGBVR comparison, taking account of the byte address selection, hits for the first unit of the instruction.

In v6 Debug and v6.1 Debug, it is IMPLEMENTATION DEFINED whether an IVA comparison on an instruction memory unit other than the first unit, following a breakpoint miss on the first unit of the instruction, can cause a Breakpoint debug event.

For Java bytecodes, v6 Debug and v6.1 Debug specify that a BRP comparison on an operand does not generate a Breakpoint debug event. A Breakpoint debug is generated only if the BRP hits on the opcode.

For Java bytecodes the instruction memory unit is a byte, and the opcode is always the first byte of the instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- アドレス範囲マスク付きの IVA 不一致：*除外範囲付き* IVA ブレークポイントと呼びます。

注

ブレークポイントでのアドレス範囲マスクのサポートは実装定義です。

すべてのタイプの IVA ブレークポイントについて、DBGBCR の条件が満たされており、命令が実行にコミットされている場合、必要な DBGBVR 比較が、バイトアドレスの選択やアドレス範囲マスクを考慮して、命令の最初の単位にヒットした場合、BRP がブレークポイントデバッグ イベントを生成します。

DBGBCR の条件が満たされ、命令が実行にコミットされているという前提で、複数のメモリ単位で構成される命令によりブレークポイントデバッグ イベントが生成される条件を、表 C3-2 に示します。

表 C3-2 複数のメモリ単位で構成される命令についてのブレークポイントデバッグ イベントの生成

DBGBVR 比較の結果[a] 最初の単位[b]	以後の単位[b]	IVA ブレークポイントのタイプ	ブレークポイントデバッグ イベントの生成
ヒット	–	任意	はい
ミス	ヒット	通常、範囲付き、除外範囲付き	予測不能
		ステップオフ	いいえ
ミス	ミス	任意	いいえ

a. バイトアドレス選択とアドレス範囲マスクが考慮されます。
b. IVA が比較される命令の単位を意味します。

v6 デバッグと v6.1 デバッグでの命令長の影響

DBGBCR の条件が満たされており、命令が実行にコミットされている場合、必要な DBGBVR 比較が、バイトアドレス選択を考慮して、命令の最初の単位にヒットした場合、BRP がブレークポイントデバッグ イベントを生成します。

v6 デバッグと v6.1 デバッグでは、命令の最初のメモリ単位でブレークポイントミスが発生した後で、最初のもの以外のメモリ単位についての IVA 比較でブレークポイントデバッグ イベントが発生するかどうかは実装定義です。

Java バイトコードの場合、v6 デバッグと v6.1 デバックでは、オペランドに対する BRP 比較ではブレークポイントデバッグ イベントが生成されないことが指定されています。ブレークポイントデバッグは、BRP がオペコードにヒットした場合のみ生成されます。

Java バイトコードでは命令のメモリ単位はバイトで、オペコードは常に命令の最初のバイトです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

- v6 Debug does not support IVA mismatch breakpoints.
- v6.1 Debug and v6 Debug do not support address range masks on breakpoints.

IVA comparison programming examples

In all Debug architecture versions, a debugger must configure the BRP so that it matches on all bytes of the first unit of the instruction, otherwise the generation of Breakpoint debug events is UNPREDICTABLE.

Before ARMv6T2, on a processor that implements the Thumb instruction set and can take an exception between the two halfwords of a Thumb BL or BLX (immediate) instruction, a debugger must treat the two halfwords as separate instructions, and set breakpoints on both halfwords. This might require two BRPs.

Note

- To ensure compatibility across ARMv6 implementations, a debugger can always treat BL or BLX (immediate) as two instructions when debugging code on an ARMv6 processor before ARMv6T2.
- The examples that follow include setting breakpoints on ThumbEE instructions. These are supported only in ARMv7.

For example, if BRPn and BRPm are two breakpoint register pairs, then:

- On any ARMv6 or ARMv7 processor:
  - To breakpoint on a Java bytecode at address 0x8001, the debugger must set DBGBVRn to 0x8000 and DBGBCRn[8:5] to 0b0010.
  - To breakpoint on a 16-bit Thumb or ThumbEE instruction starting at address 0x8002, a debugger must set DBGBVRn to 0x8000 and DBGBCRn[8:5] to 0b1100.
  - To breakpoint on an ARM instruction starting at address 0x8004, a debugger must set DBGBVRn to 0x8004 and DBGBCRn[8:5] to 0b1111.
- On an ARMv7 or ARMv6T2 processor, a debugger sets breakpoints on a 32-bit Thumb instruction, or on a 16-bit or a 32-bit ThumbEE instruction, in exactly the same way as on a 16-bit Thumb instruction. For example:
  - To breakpoint on a 16-bit or a 32-bit Thumb or ThumbEE instruction starting at address 0x8000, the debugger must set DBGBVRn to 0x8000 and DBGBCRn[8:5] to 0b0011. These are the settings for breakpointing on any Thumb or ThumbEE instruction, including BL and BLX (immediate).
- On an ARMv6 or ARMv6K processor:
  - To breakpoint on a Thumb BL or BLX instruction at address 0x8000, a debugger must set DBGBVRn to 0x8000, and DBGBCRn[8:5] to 0b1111.
  - To breakpoint on a Thumb BL or BLX instruction at address 0x8002, a debugger must set DBGBVRn to 0x8000, DBGBVRm to 0x8004, DBGBCRn[8:5] to 0b1100, and DBGBCRm[8:5] to 0b0011.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

- v6 デバッグでは、IVA 不一致のブレークポイントはサポートされていません。
- v6.1 デバッグと v6 デバッグでは、ブレークポイントへのアドレス範囲マスクはサポートされていません。

IVA 比較のプログラミング例

デバッグアーキテクチャのすべてのバージョンで、デバッガは、命令の最初の単位に属するすべてのバイトに一致するように BRP を構成する必要があります。これを行わない場合、ブレークポイントデバッグ イベントの生成は予測不能です。

ARMv6T2 以前では、プロセッサに Thumb 命令セットを実装し、Thumb の BL または BLX（イミディエート）命令の 2 つのハーフワードの間で例外を取得することができます。このようなプロセッサでは、デバッガが、これら 2 つのハーフワードを別の命令として扱い、両方のハーフワードにブレークポイントを設定する必要があります。このために 2 つの BRP が必要となることがあります。

注

- ARMv6 実装間にわたって互換性を保証するため、ARMv6T2 以前の ARMv6 プロセッサでコードのデバッグを行う場合、デバッガで BL または BLX（イミディエート）を常に 2 つの命令として扱うことができます。
- 以下に示す例には、ThumbEE 命令へのブレークポイントの設定が含まれています。この機能は、ARMv7 でのみサポートされています。

例として、BRPn と BRPm が 2 つのブレークポイント レジスタペアであるとすると、次の規則が適用されます。

- 任意の ARMv6 または ARMv7 プロセッサの場合
  - アドレス 0x8001 の Java バイトコードにブレークポイントを設定するには、デバッガは DBGBVRn を 0x8000 に、DBGBCRn[8:5] を 0b0010 に設定する必要があります。
  - アドレス 0x8002 から始まる 16 ビット Thumb または ThumbEE 命令にブレークポイントを設定するには、デバッガは DBGBVRn を 0x8000 に、DBGBCRn[8:5] を 0b1100 に設定する必要があります。
  - アドレス 0x8004 から始まる ARM 命令にブレークポイントを設定するには、デバッガは DBGBVRn を 0x8004 に、DBGBCRn[8:5] を 0b1111 に設定する必要があります。
- ARMv7 または ARMv6T2 プロセッサでは、デバッガが 32 ビット Thumb 命令や、16 ビットまたは 32 ビット ThumbEE 命令にブレークポイントを設定する方法は、16 ビット Thumb 命令の場合と全く同じです。次に例を示します。
  - アドレス 0x8000 から始まる 16 ビットまたは 32 ビットの Thumb または ThumbEE 命令にブレークポイントを設定するには、デバッガは DBGBVRn を 0x8000 に、DBGBCRn[8:5] を 0b0011 に設定する必要があります。これらは、BL および BLX（イミディエート）を含む、すべての Thumb または ThumbEE 命令にブレークポイントを設定する場合に適用されます。
- ARMv6 または ARMv6K プロセッサの場合
  - アドレス 0x8000 にある Thumb の BL または BLX 命令にブレークポイントを設定するには、デバッガは DBGBVRn を 0x8000 に、DBGBCRn[8:5] を 0b1111 に設定する必要があります。
  - アドレス 0x8002 にある Thumb の BL または BLX 命令にブレークポイントを設定するには、デバッガは DBGBVRn を 0x8000、DBGBVRm を 0x8004、DBGBCRn[8:5] を 0b1100、DBGBCRm[8:5] を 0b0011 に設定する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

When programming DBGBVR for IVA match or mismatch, the debugger must program DBGBVR[1:0] to 0b00, otherwise Breakpoint debug event generation is UNPREDICTABLE.

Context ID comparisons for Debug event generation

A Context ID comparison depends on the value in the DBGBVR matching the Context ID, held in the Context ID Register, when the instruction is committed for execution. The breakpoint types that depend on a Context ID comparison are:

- Unlinked Context ID match
- Linked Context ID match.

When the DBGBCR is programmed for one of these debug types, the debug logic generates a Breakpoint debug event only if all the other conditions for the breakpoint are met, and the Context ID equals the value in the DBGBVR.

In the linked case, the BRP that is programmed for a Context ID match is linked to at least one of:

- a BRP programmed for Linked IVA match or mismatch
- a Watchpoint Register Pair (WRP) programmed for linked Data Virtual Address (DVA) match.

In the linked IVA cases, the debug logic generates a Breakpoint debug event only if all the other conditions for the breakpoint are met, the Context ID comparison is successful, and the IVA comparison in the linked BRP is successful, see *IVA comparisons for Debug event generation* on page C3-8. See *Additional conditions for linked BRPs* on page C3-15 for more information.

In the linked DVA case, the debug logic generates a Watchpoint debug event only if all the other conditions for the watchpoint are met, the Context ID comparison is successful, and the DVA comparison in the linked WRP is successful, See *Watchpoint debug events* on page C3-15 for more information.

Note

- You cannot define a Breakpoint debug event based on a Context ID mismatch.
- You can link a BRP programmed for linked Context ID match to any number of:
  - BRPs programmed for Linked IVA match or mismatch
  - WRPs programmed for Linked DVA match.

  This means you can use a single BRP to define the Context ID match for multiple breakpoints and watchpoints.

Additional considerations for IVA mismatch breakpoints

The following subsections describe additional considerations for IVA mismatch breakpoints:

- *Interaction of IVA mismatch breakpoints with other breakpoints and Vector Catch* on page C3-14
- *Generation of IVA mismatch breakpoints on branch to self instructions* on page C3-14.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

DBGBVR を IVA 一致または不一致にプログラミングする場合、デバッガは DBGBVR[1:0] を 0b00 にプログラムする必要があります。これを行わない場合、ブレークポイントデバッグ イベントの生成は予測不能です。

デバックイベント生成のためのコンテキスト ID 比較

コンテキスト ID 比較は、命令が実行にコミットされた時点で、DBGBVR の値が、コンテキスト ID レジスタに保持されているコンテキスト ID と一致するかどうかのチェックです。コンテキスト ID 比較に依存するブレークポイントは、次のタイプのものです。

- リンクなしのコンテキスト ID 一致
- リンク付きのコンテキスト ID 一致

これらのデバッグタイプのうち 1 つについて DBGBCR がプログラムされている場合、ブレークポイントの他の条件がすべて満たされ、コンテキスト ID が DBGBVR の値と等しい場合のみ、デバッグロジックがブレークポイントデバッグ イベントを生成します。

リンク付きの場合、コンテキスト ID 一致用にプログラムされている BRP は、次のうち最低 1 つとリンクされています。

- リンク付き IVA 一致または不一致用にプログラムされている BRP
- リンク付きデータ仮想アドレス (DVA) 一致用にプログラムされているウォッチポイント レジスタペア (WRP)

リンク付き IVA の場合は、ブレークポイントに関する他の条件がすべて満たされ、コンテキスト ID 比較が成功し、リンク付き BRP で IVA 比較が成功した場合のみ、デバッグロジックがブレークポイント デバッグ イベントを生成します。*デバックイベント生成のための IVA 比較* P. C3-8 を参照して下さい。詳細については、*リンク付き BRP についての追加条件* P. C3-15 を参照して下さい。

リンク付き DVA の場合は、ウォッチポイントに関する他の条件がすべて満たされ、コンテキスト ID 比較が成功し、リンク付き WRP で DVA 比較が成功した場合のみ、デバッグロジックがウォッチポイントデバッグ イベントを生成します。詳細については、*ウォッチポイントデバッグ イベント* P. C3-15 を参照して下さい。

注

- コンテキスト ID 不一致に基づいてブレークポイントデバッグ イベントを定義することはできません。
- リンク付きコンテキスト ID 一致用にプログラムされた BRP は、次に示す対象に、数の制限なくリンクできます。
  - リンク付き IVA 一致または不一致用にプログラムされている BRP
  - リンク付き DVA 一致用にプログラムされている WRP

  つまり、単一の BRP を使用して、複数のブレークポイントおよびウォッチポイントのコンテキスト ID 一致を定義できます。

IVA 不一致ブレークポイントについての追加の考慮事項

次に示すサブセクションでは、IVA 不一致ブレークポイントについての追加の考慮事項について説明しています。

- *IVA 不一致ブレークポイントと、他のブレークポイントおよびベクタキャッチとの相互動作* P. C3-14
- *自分に分岐する命令での IVA 不一致ブレークポイントの生成* P. C3-14

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Interaction of IVA mismatch breakpoints with other breakpoints and Vector Catch

When a BRPn is programmed for IVA mismatch and does not generate a Breakpoint debug event because the general conditions specified in DBGBCRn are not met, this does not affect the generation of:

- Breakpoint debug events by other BRPs
- Vector Catch debug events.

Note

In this context, the general conditions specified in DBGBCR not being met means that at least one of the following applies:

- the BRP is not enabled
- the Privileged mode control bits of the DBGBCR do not match the mode of the processor
- DBGBCR is configured for linked Context ID matching but the linked BRP either is not enabled or does not match the current Context ID
- the Security Extensions are implemented, and the Security state control field of DBGBCR does not match the security state of the processor.

However, if the general conditions specified in DBGBCRn are met, and BRPn does not generate a Breakpoint debug event only because the IVA fails the comparison required for an IVA mismatch, then the failure of this comparison can affect the generation of other debug events:

- if any other BRP, BRPm, hits on its required comparison with the IVA and meets the general conditions specified in DBGBCRm, it is UNPREDICTABLE whether BRPm generates a Breakpoint debug event
- if the Vector Catch Register defines a Vector Catch that matches the IVA, it is UNPREDICTABLE whether a Vector Catch debug event is generated.

Generation of IVA mismatch breakpoints on branch to self instructions

This section describes the generation of Breakpoint debug events when the IVA of an instruction that branches to itself misses a BRP programmed for IVA mismatch, and all the general conditions specified in the DBGBCR are met. See the IVA mismatch column of Table C10-11 on page C10-56 for details of when an IVA mismatch comparison misses. In this case:

1. The first time the instruction is committed for execution the BRP does not generate a Breakpoint debug event.
2. Because the instruction branches to itself, if no exception is generated, the instruction is committed for execution again. On this and any subsequent execution, it is UNPREDICTABLE whether the BRP generates a Breakpoint debug event.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

IVA 不一致ブレークポイントと、他のブレークポイントおよびベクタキャッチとの相互動作

BRPn が IVA 不一致用にプログラムされており、DBGBCRn に指定されている一般的な条件が満たされていないためにブレークポイントデバッグ イベントが生成されない場合、この動作は次のイベントの生成には影響しません。

- 他の BRP によるブレークポイントデバッグ イベント
- ベクタキャッチデバッグ イベント

注

この文脈では、DBGBCR に指定されている一般的な条件が満たされないとは、次の条件のうち最低 1 つが当てはまることを意味します。

- BRP が非稼動である
- DBGBCR の特権モード制御ビットが、プロセッサのモードと一致しない
- DBGBCR がリンク付きコンテキスト ID 一致用に構成されているが、リンク付き BRP が稼働中でないか、現在のコンテキスト ID と一致しない
- セキュリティ拡張機能が実装されており、DBGBCR のセキュリティ状態制御フィールドが、プロセッサのセキュリティ状態と一致しない

ただし、DBGBCRn で指定されている一般的な条件が満たされており、IVA 不一致で要求される比較が IVA で失敗しただけの理由から、BRPn がブレークポイントデバッグ イベントを生成しない場合は、この比較の失敗は他のデバッグイベントの生成に次のように影響します。

- 他の BRP である BRPm が、要求される IVA との比較にヒットし、DBGBCRm で指定されている一般的な条件を満たしている場合、BRPm がブレークポイントデバッグ イベントを生成するかどうかは予測不能です。
- ベクタキャッチ レジスタが、IVA と一致するベクタキャッチを定義している場合、ベクタキャッチデバッグ イベントが生成されるかどうかは予測不能です。

自分に分岐する命令での IVA 不一致ブレークポイントの生成

このセクションでは、自分に分岐する命令の IVA が、IVA 不一致用にプログラムされている BRP にミスし、DBGBCR で指定されているすべての一般的な条件が満たされている場合の、ブレークポイントデバッグ イベントの生成について説明します。IVA 不一致の比較がミスする条件の詳細については、P. C10-56 の表 C10-11 にある IVA 不一致の列を参照して下さい。この場合、次の動作が行われます。

1. 命令が最初に実行にコミットされたときには、BRP はブレークポイントデバッグ イベントを生成しません。
2. この命令は自分に分岐するため、例外が生成されなければ、命令は再度実行にコミットされます。このとき、およびそれ以後の実行では、BRP がブレークポイントデバッグ イベントを生成するかどうかは予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

Instructions that branch to themselves include:

- a branch instruction that specifies itself as the branch destination
- a load instruction that loads the PC from a memory location that holds the address of that load instruction.

Additional conditions for linked BRPs

When you link two BRPs to define a single Linked IVA match or mismatch breakpoint, if BRPn defines the IVA match or mismatch and BRPm defines the Context ID match:

- for the DBGBCR fields described in *Debug event generation conditions defined by the DBGBCR* on page C3-7, you must program DBGBCRn and DBGBCRm as follows:
  - in DBGBCRn, program the Security state control and Privileged mode control fields to define the required conditions for Debug event generation
  - in DBGBCRm, program the Security state control field to 0b00, and the Privileged mode control field to 0b11
- you must program the Linked BRP number field:
  - of DBGBCRn with the value of m
  - of DBGBCRm to zero
- you must program the DBGBVR meaning field:
  - of DBGBCRn for Linked IVA match or mismatch
  - of DBGBCRm for Linked Context ID match
- BRPm must support Context ID comparisons.

Breakpoint debug event generation is UNPREDICTABLE if you do not meet all these conditions.

You must also set the Breakpoint enable bits in DBGBCRn and DBGBCRm to 1, to enable both BRPs.

Note

If you fail to enable either or both of the BRPs, BRPn never generates any Breakpoint debug events.

For more information see *Linked comparisons* on page C10-59

C3.2.2 Watchpoint debug events

A Watchpoint debug event is defined by a pair of registers described as a *Watchpoint Register Pair* (WRP), comprising a *Watchpoint Control Register* (DBGWCR) and a *Watchpoint Value Register* (DBGWVR). WRPs, DBGWCRs, and DBGWVRs number upwards from 0, with WRPn comprising DBGWCRn and DBGWVRn. For details of the Watchpoint registers see:

- *Watchpoint Control Registers (DBGWCR)* on page C10-61

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

自分に分岐する命令には、次のものがあります。

- 分岐先として自分自身を指定している分岐命令
- ロード命令が、そのロード命令のアドレスを保持しているメモリ位置から PC をロードする場合

リンク付き BRP についての追加条件

2 つの BRP をリンクして、単一のリンク付き IVA 一致または不一致ブレークポイントを定義するとき、BRPn が IVA 一致または不一致を定義し、BRPm がコンテキスト ID 一致を定義する場合、次の条件が適用されます。

- *DBGBCR により定義されるデバッグイベント生成条件* P. C3-7 に説明されている DBGBCR フィールドについて、DBGBCRn と DBGBCRm を次のようにプログラムする必要があります。
  - DBGBCRn で、セキュリティ状態制御フィールドと特権モード制御フィールドをプログラムして、デバッグイベントの生成に必要な条件を定義します。
  - DBGBCRm で、セキュリティ状態制御フィールドを 0b00、特権モード制御フィールドを 0b11 にプログラムします。
- リンク付き BRP 番号フィールドは、次のようにプログラムする必要があります。
  - DBGBCRn では、m の値をプログラムする
  - DBGBCRm では、0 に設定する
- DBGBVR の意味フィールドは、次のようにプログラムする必要があります。
  - DBGBCRn では、リンク付き IVA 一致または不一致用にプログラムする
  - DBGBCRm では、リンク付きコンテキスト ID 一致用にプログラムする
- BRPm は、コンテキスト ID 比較をサポートしている必要があります。

これらの条件のすべてが満たされていない場合、ブレークポイントデバッグイベントの生成は予測不能です。

また、DBGBCRn および DBGBCRm のブレークポイント イネーブルビットを 1 にセットし、両方の BRP を稼動する必要があります。

注

BRP のいずれかまたは両方を稼動していない場合、BRPn でブレークポイントデバッグ イベントは一切生成されません。

詳細については、*リンク付き比較* P. C10-59 を参照して下さい。

C3.2.2 ウォッチポイントデバッグ イベント

ウォッチポイントデバッグ イベントは、*ウォッチポイント制御レジスタ*(DBGWCR) と *ウォッチポイント値レジスタ*(DBGWVR) によって定義されます。これらのレジスタは総称として *ウォッチポイント レジスタペア*(WRP) と呼ばれます。WRP、DBGWCR、DBGWVR の番号は 0 以上で、WRPn は DBGWCRn と DBGWVRn で構成されます。ウォッチポイントレジスタの詳細については、次のセクションを参照して下さい。

- *ウォッチポイント制御レジスタ (DBGWCR)* P. C10-61

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- *Watchpoint Value Registers (DBGWVR)* on page C10-60.

The DBGDIDR.WRPs field specifies the number of WRPs implemented, see *Debug ID Register (DBGDIDR)* on page C10-3.

A WRP can be linked to a BRP, to define a single watchpoint event. The WRP holds a virtual address for comparison, and the BRP holds a Context ID value. For more information, see *Linked comparisons* on page C10-59.

A Watchpoint debug event is defined based on comparisons of a *Data Virtual Address* (DVA) with the value held in a WVR. See *Memory addresses* on page C3-23 for the definition of a DVA.

For a given Watchpoint Register Pair, WRPn, a Watchpoint debug event occurs when all of the following are true:

- The watchpoint is enabled, in DBGWCRn.
- The DVA matches the value in DBGWVRn.
- When the processor tests the WRP, all the conditions of DBGWCRn are met.
- If linking is enabled in DBGWCRn, when the processor tests the WRP, the Linked Context ID matching BRP, BRPm, meets the following conditions:
  - — the BRP is enabled, in DBGBCRm
  - — the value held in the DBGBVRm matches the Context ID held in the CONTEXTIDR.

  For more information about BRPs see *Breakpoint debug events* on page C3-5.
- The instruction that initiated the memory access is committed for execution. A Watchpoint debug event is generated only if the instruction passes its condition code check.

Note

A watchpoint match does not require the access to match exactly the watched address. A match is generated on any access to any watched byte or bytes. For example, a match is generated on an unaligned word access that includes a byte that is being watched, even when the watched byte is not in the same word as the start address of the unaligned word.

All instructions that are defined as memory access instructions can generate Watchpoint debug events. For information about which instructions are memory accesses see *Alphabetical list of instructions* on page A8-14. Watchpoint debug event generation can be conditional on whether the memory access is a load access or a store access.

For a Store-Exclusive instruction, if the target address of the instruction would generate a Watchpoint debug event, but the check of whether the Store-Exclusive operation has control of the exclusive monitors returns FALSE, then it is IMPLEMENTATION DEFINED whether the processor generates the Watchpoint debug event.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- *ウォッチポイント値レジスタ (DBGWVR)* P. C10-60

DBGDIDR.WRP のフィールドは、実装されている WRP の数を指定します。*デバッグ ID レジスタ (DBGDIDR)* P. C10-3 を参照して下さい。

WRP を BRP にリンクして、単一のウォッチポイントイベントを定義することができます。WRP には比較用の仮想アドレスが、BRP にはコンテキスト ID の値が保持されます。詳細については、*リンク付き比較* P. C10-59 を参照して下さい。

ウォッチポイントデバッグ イベントは、*データ仮想アドレス*(DVA) と WVR に保持されている値との比較に基づいて定義されます。DVA の定義については、*メモリアドレス* P. C3-23 を参照して下さい。

与えられたウォッチポイント レジスタペア WRPn について、次の条件のすべてが TRUE の場合にウォッチポイントデバッグ イベントが発生します。

- DBGWCRn でウォッチポイントが可能である。
- DVA が DBGWVRn の値と一致している。
- プロセッサが WRP をテストしたとき、DBGWCRn のすべての条件が満たされている。
- DBGWCRn でリンクが可能な場合、プロセッサが WRP をテストしたとき、リンク付きコンテキスト ID 一致 BRP である BRPm が次の条件を満たしている。
  - – DBGBCRm で BRP が稼動している。
  - – DBGBVRm に保持されている値が、CONTEXTIDR に保持されているコンテキスト ID と一致している。

  BRP の詳細については、*ブレークポイントデバッグ イベント* P. C3-5 を参照して下さい。
- メモリアクセスを開始した命令が実行にコミットされている。ウォッチポイントデバッグ イベントは、命令が条件コードチェックにパスした場合のみ生成されます。

注

ウォッチポイント一致には、アクセスがウォッチ対象のアドレスと正確に一致する必要はありません。ウォッチ対象の任意のバイトに対する任意のアクセスによって一致が生成されます。例えば、ウォッチ対象のバイトを含むアンアラインドのワードアクセスによって一致が生成されます。ウォッチ対象のバイトが、アンアラインドワードの開始アドレスと同じワードに含まれていない場合もこの動作が適用されます。

メモリアクセス命令として定義されているすべての命令は、ウォッチポイントデバッグ イベントを生成する可能性があります。どの命令がメモリアクセス命令であるかの詳細については、*命令のアルファベット順リスト* P. A8-14 を参照して下さい。ウォッチポイントデバッグ イベントの生成は、メモリアクセスがロードアクセスか、ストアアクセスかによって条件付きとすることができます。

排他ストア命令の場合、命令のターゲットアドレスがウォッチポイントデバッグ イベントを生成するものであるが、排他ストア操作が排他モニタを制御を与えられているかのチェックで FALSE が返された場合、プロセッサがウォッチポイントデバッグ イベントを生成するかどうかは実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For each of the memory hint instructions, PLD and PLI, it is IMPLEMENTATION DEFINED whether the instruction generates Watchpoint debug events. If either or both of the PLD and PLI instructions normally generates Watchpoint debug events, the behavior must be:

- For the PLI instruction:
  - — no watchpoint is generated in a situation where, if the instruction was a real fetch rather than a hint, the real fetch would generate a Prefetch Abort exception
  - — in all other situations a Watchpoint debug event is generated.
- For the PLD instruction:
  - — no watchpoint is generated in a situation where, if the instruction was a real memory access rather than a hint, the real memory access would generate a Data Abort exception
  - — in all other situations a Watchpoint debug event is generated.
- When watchpoint generation is conditional on the type of memory access, a memory hint instruction is treated as generating a load access.

It is IMPLEMENTATION DEFINED whether the following cache maintenance operations generate Watchpoint debug events:

- Clean data or unified cache line by MVA to PoU, DCCMVAU
- Clean data or unified cache line by MVA to PoC, DCCMVAC
- Invalidate data or unified cache line by MVA to PoC, DCIMVAC
- Invalidate instruction cache line by MVA to PoU, ICIMVAU
- Clean and Invalidate data or unified cache line by MVA to PoC, DCCIMVAC.

When Watchpoint debug event generation by these cache maintenance operations is implemented, the behavior must be:

- the cache maintenance operation must generate a Watchpoint debug event on a DVA match, regardless of whether the data is stored in any cache
- when watchpoint generation is conditional on the type of memory access, a cache maintenance operation is treated as generating a store access.

For regular data accesses, the size of the access is considered when determining whether a watched byte is being accessed. The size of the access is IMPLEMENTATION DEFINED for:

- memory hint instructions, PLD and PLI
- cache maintenance operations.

Watchpoint debug events are precise and can be *synchronous* or *asynchronous*:

- a synchronous Watchpoint debug event acts like a synchronous abort exception on the memory access instruction itself
- an asynchronous Watchpoint debug event acts like a precise asynchronous abort exception that cancels a later instruction.

For more information, see *Synchronous and Asynchronous Watchpoint debug events* on page C3-18.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

メモリヒント命令 PLD および PLI のそれぞれについて、その命令がウォッチポイントデバッグ イベントを生成するかどうかは実装定義です。PLD および PLI 命令のいずれかまたは両方が、通常はウォッチポイントデバッグ イベントを生成する場合、次のような動作が行われる必要があります。

- PLI 命令の場合
  - – 命令がヒントではなく実際のフェッチであったと仮定した場合に、実際のフェッチでプリフェッチアボート例外が生成される状況では、ウォッチポイントは生成されません。
  - – 他のすべての状況では、ウォッチポイントデバッグ イベントが生成されます。
- PLD 命令の場合
  - – 命令がヒントではなく実際のメモリアクセスであったと仮定した場合に、実際のメモリアクセスでデータアボート例外が生成される状況では、ウォッチポイントは生成されません。
  - – 他のすべての状況では、ウォッチポイントデバッグ イベントが生成されます。
- ウォッチポイントの生成がメモリアクセスのタイプに関する条件付きの場合、メモリヒント命令はロードアクセスの生成とみなされます。

次のキャッシュ保守操作でウォッチポイントデバッグ イベントが生成されるかどうかは実装定義です。

- MVA による、PoU までのデータまたは統一キャッシュラインのクリーニング、DCCMVAU
- MVA による、PoC までのデータまたは統一キャッシュラインのクリーニング、DCCMVAC
- MVA による、PoC までのデータまたは統一キャッシュラインの無効化、DCIMVAC
- MVA による、PoU までの命令キャッシュラインの無効化、ICIMVAU
- MVA による、PoC までのデータまたは統一キャッシュラインのクリーニングと無効化、DCCIMVAC

これらのキャッシュ保守操作によるウォッチポイントデバッグ イベント生成が実装されている場合、次のように動作する必要があります。

- キャッシュ保守操作は、データがいずれかのキャッシュに格納されているかどうかにかかわらず、DVA 一致時にウォッチポイントデバッグ イベントを生成する必要があります。
- ウォッチポイントの生成がメモリアクセスのタイプに関する条件付きの場合、キャッシュ保守操作はストアアクセスの生成とみなされます。

通常のデータアクセスの場合、ウォッチ対象のバイトがアクセスされているかどうかを判定するときに、アクセスのサイズが考慮されます。次の場合、アクセスのサイズは実装定義です。

- メモリヒント命令の PLD および PLI
- キャッシュ保守操作

ウォッチポイントデバッグ イベントは正確で、*同期*または*非同期*です。

- 同期ウォッチポイントデバッグ イベントは、メモリアクセス命令自体に対しての同期アボート例外と同様に動作します。
- 非同期ウォッチポイントデバッグ イベントは、以後の命令をキャンセルする正確な非同期アボート例外と同様に動作します。

詳細については、*同期と非同期のウォッチポイントデバッグ イベント* P. C3-18 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For the ordering of debug events, ARMv7 requires that:

- Regardless of the actual ordering of memory accesses, Watchpoint debug events must be taken in program order. See *Debug event prioritization* on page C3-43.
- Watchpoint debug events must behave as if the processor tested for any possible Watchpoint debug event before the memory access was observed, regardless of whether the Watchpoint debug event is synchronous or asynchronous. See *Generation of debug events* on page C3-40.

Synchronous and Asynchronous Watchpoint debug events

ARMv7 permits watchpoints to be either *synchronous* or *asynchronous*. An implementation can implement synchronous watchpoints, asynchronous watchpoints, or both. It is IMPLEMENTATION DEFINED under what circumstances a watchpoint is synchronous or asynchronous.

ARMv6 only permits asynchronous watchpoints.

Synchronous Watchpoint debug events

A synchronous Watchpoint debug event acts like a synchronous abort:

- The debug event occurs before any following instructions or exceptions have altered the state of the processor.
- The value in the base register for the memory access is not updated.

  Note

  The Base Updated Abort Model is not permitted in ARMv7.

- If the instruction was a register load, the data returned is marked as invalid and:
  - if the instruction was a single register load, the destination is not updated
  - if the instruction loaded multiple registers, the values in the destination registers, other than the PC and base register, are UNKNOWN.
- If the instruction is a coprocessor load, the values left in the coprocessor registers are UNKNOWN.
- If the instruction is a store, the content of the memory location written to is unchanged.

When invasive debug is enabled and Monitor debug-mode is selected, if Watchpoint debug events are permitted a synchronous Watchpoint debug event generates a synchronous Data Abort exception. On a synchronous Watchpoint debug event, the DBGDSCR.MOE field is set to Synchronous Watchpoint occurred.

When an instruction that causes multiple memory operations is addressing Device or Strongly-ordered memory, if a synchronous Watchpoint debug event is signaled by a memory operation other than the first operation of the instruction, the memory access rules might not be maintained. Examples of instructions that cause multiple memory operations are the `LDM` and `LDC` instructions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

デバッグイベントの順序について、ARMv7 では次の規則に従う必要があります。

- メモリアクセスの実際の順序にかかわらず、ウォッチポイントデバッグ イベントはプログラム上の順序で取得される必要があります。詳細については、*デバッグイベントの優先度付け* P. C3- 43 を参照して下さい。
- ウォッチポイントデバッグ イベントは、同期か非同期かにかかわらず、メモリアクセスが観測される前に、プロセッサが可能なウォッチポイントデバッグ イベントをすべてチェックしたのと同様に動作する必要があります。*デバッグ イベントの生成* P. C3- 40 を参照して下さい。

同期と非同期のウォッチポイントデバッグ イベント

ARMv7 では、ウォッチポイントは*同期*と*非同期*の両方が許可されます。実装では、同期ウォッチポイントと非同期ウォッチポイントのいずれかを実装することも、両方を実装することもできます。どのような状況でウォッチポイントが同期または非同期であるかは実装定義です。

ARMv6 では非同期ウォッチポイントのみが許可されます。

同期ウォッチポイントデバッグ イベント

同期ウォッチポイントデバッグ イベントは、同期アボートと同様に動作します。

- デバッグイベントは、以後の命令や例外によってプロセッサの状態が変更される前に発生します。
- メモリアクセスについて、ベースレジスタの値は更新されません。

  注

  ベース更新アボートモデルは、ARMv7 では許可されません。

- 命令がレジスタロードの場合、返されるデータは無効にマークされ、次の条件が適用されます。
  - 命令が単一レジスタロードの場合、デスティネーションは更新されません。
  - 命令により複数のレジスタがロードされた場合、PC およびベースレジスタを除くデスティネーションレジスタの値は未知です。
- 命令がコプロセッサロードの場合、コプロセッサレジスタに残される値は未知です。
- 命令がストアの場合、書き込まれるメモリ位置の内容は変更されません。

侵襲性デバッグが可能で、モニタデバッグ モードが選択されている場合、ウォッチポイントデバッグ イベントが許可されていれば、同期ウォッチポイントデバッグ イベントは同期データアボート例外を生成します。同期ウォッチポイントデバッグ イベントが発生すると、DBGDSCR.MOE フィールドは、発生した同期ウォッチポイントに設定されます。

複数のメモリ操作を引き起こす命令がデバイスまたはストロングリオーダ メモリをアドレス指定している場合、命令の最初の操作以外のメモリ操作によって同期ウォッチポイントデバッグ イベントが発生すると、メモリアクセスの規則が維持されない可能性があります。複数のメモリ操作を引き起こす命令の例として、`LDM` および `LDC` 命令があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For example, if the second memory operation of an STM instruction signals a synchronous Watchpoint debug event, then when the instruction is re-tried following processing of the debug event, the first memory operation is repeated. This behavior is not normally permitted for accesses to Device or Strongly-ordered memory.

To avoid this circumstance, debuggers must not set watchpoints on addresses in regions of Device or Strongly-ordered memory that might be accessed in this way. The address range masking features of watchpoints can be used to set a watchpoint on an entire region, ensuring the synchronous Watchpoint debug event is taken on the first operation of such an instruction.

Asynchronous Watchpoint debug events

An asynchronous Watchpoint debug event acts like a precise asynchronous abort. Its behavior is:

- The watchpointed instruction *must* have completed, and other instructions that followed it, in program order, might have completed. For more information, see *Recognizing asynchronous Watchpoint debug events*.
- The watchpoint *must* be taken before any exceptions that occur in program order after the watchpoint is triggered.
- All the registers written by the watchpointed instruction are updated.
- Any memory accessed by the watchpointed instruction is updated.

When invasive debug is enabled and Monitor debug-mode is selected, if Watchpoint debug events are permitted an asynchronous Watchpoint debug event generates a precise asynchronous Data Abort exception.

An asynchronous Watchpoint debug event is not an abort and is not affected by architectural rules about aborts, including the rules about external aborts and asynchronous aborts. An asynchronous Watchpoint debug event:

- is not affected by the SCR.EA bit
- is not ignored when the CPSR.A bit is set to 1.

On an asynchronous Watchpoint debug event, the DBGDSCR.MOE field is set to Asynchronous Watchpoint occurred.

Recognizing asynchronous Watchpoint debug events

When an instruction that consists of multiple memory operations is accessing Device or Strongly-ordered memory, and an asynchronous Watchpoint debug event is signaled by a memory operation other than the first operation of the instruction, the debug event *must not* cause Debug state entry or a debug exception until all the operations have completed. This ensures the memory access rules for Device and Strongly-ordered memory are preserved.

Examples of instructions that cause multiple memory operations are the LDM and LDC instructions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

例えば、STM 命令の 2 番目のメモリ操作によって同期ウォッチポイントデバッグ イベントが発生する場合、デバッグイベントの以後の処理で命令が再試行されると、最初のメモリ操作が繰り返されます。デバイスまたはストロングリオーダ メモリへのアクセスの場合、通常はこの動作は許可されません。

この状況を回避するには、デバッガは、このような方式でアクセスされる可能性があるデバイスまたはストロングリオーダ メモリ領域のアドレスに、ウォッチポイントを設定しないようにする必要があります。ウォッチポイントのアドレス範囲マスク機能を使用して領域全体にウォッチポイントを設定すると、このような命令の最初の動作で同期ウォッチポイントデバッグ イベントが取得されることを保証できます。

非同期ウォッチポイントデバッグ イベント

非同期ウォッチポイントデバッグ イベントは、正確な非同期アボートと同様に動作します。この動作は次のとおりです。

- ウォッチポイントが設定された命令は完了している*必要があります*。その後に続く他の命令は、プログラムの順序で完了している可能性があります。詳細については、*非同期ウォッチポイントデバッグ イベントの認識*を参照して下さい。
- ウォッチポイントは、プログラム順序でそれがトリガされた後に発生する、いかなる例外よりも先に取得される*必要があります*。
- ウォッチポイントが設定された命令によって書き込まれるすべてのレジスタは更新されます。
- ウォッチポイントが設定された命令によってアクセスされるメモリがあれば、すべて更新されます。

侵襲性デバッグが可能で、モニタデバッグ モードが選択されている場合、ウォッチポイントデバッグ イベントが許可されていれば、非同期ウォッチポイントデバッグ イベントは正確な非同期データアボート例外を生成します。

非同期ウォッチポイントデバッグ イベントはアボートではなく、外部アボートおよび非同期アボートに関する規則も含めて、アボートに関するアーキテクチャの規則の影響を受けません。非同期ウォッチポイントデバッグ イベントには、次の規則が適用されます。

- SCR.EA ビットの影響を受けません。
- CPSR.A ビットが 1 にセットされているときに無視されません。

非同期ウォッチポイントデバッグ イベントが発生すると、DBGDSCR.MOE フィールドは、発生した非同期ウォッチポイントに設定されます。

非同期ウォッチポイントデバッグ イベントの認識

複数のメモリ操作で構成される命令がデバイスまたはストロングリオーダ メモリにアクセスし、命令の最初の操作以外のメモリ操作によって非同期ウォッチポイントデバッグ イベントが引き起こされた場合、すべての操作が完了するまで、デバッグイベントでデバック状態への遷移やデバッグ例外を引き起こすことは*できません*。これによって、デバイスおよびストロングリオーダ メモリに関するメモリアクセス規則の維持が保証されます。

複数のメモリ操作を引き起こす命令の例として、LDM および LDC 命令があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

To understand why the architecture does not permit the asynchronous Watchpoint debug event to be taken before the watchpointed instruction completes, consider an LDM instruction accessing Device or Strongly-ordered memory, with an asynchronous Watchpoint debug event signaled after the first word of memory is accessed. If the debug event was taken immediately, the LDM would be re-executed on return from the event handler. This would cause a new access to the first word of memory, breaking the rule that, for Device or Strongly-ordered memory, each memory operation of an instruction is issued precisely once.

C3.2.3 BKPT Instruction debug events

A BKPT Instruction debug event occurs when a BKPT instruction is committed for execution. BKPT is an unconditional instruction.

BKPT Instruction debug events are synchronous. That is, the debug event acts like an exception that cancels the BKPT instruction.

For details of the BKPT instruction and its encodings in the ARM and Thumb instruction sets see *BKPT* on page A8-56.

C3.2.4 Vector Catch debug events

The Vector Catch Register (DBGVCR) controls Vector Catch debug events, see *Vector Catch Register (DBGVCR)* on page C10-67.

A Vector Catch debug event occurs when:

- The IVA of an instruction matches a vector address for the current security state.

  See *Memory addresses* on page C3-23 for a definition of the IVA.
- When the processor tests for the possible vector catch, the corresponding bit of the DBGVCR is set to 1, indicating that vector catch is enabled.
- The instruction is committed for execution. The debug event is generated whether the instruction passes or fails its condition code check.

If all the conditions for a Vector Catch debug event are met, the processor generates the event regardless of the mode in which it is executing.

The processor must test for any possible Vector Catch debug events before it executes the instruction.

If the Security Extensions are not implemented the debug logic uses only one set of vector addresses to generate Vector Catch debug events, and these are called the *Local vector addresses*.

If the Security Extensions are implemented, the debug logic uses three sets of vector addresses to generate Vector Catch debug events:

- One set for exceptions taken in the Non-secure exception modes. These are called the *Non-secure Local vector addresses*.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

アーキテクチャが、ウォッチポイントが設定された命令の完了前に非同期ウォッチポイントデバッグ イベントの取得を許可しない理由を理解するには、LDM 命令がデバイスまたはストロングリオーダ メモリにアクセスし、メモリの最初のワードがアクセスされた後で非同期ウォッチポイントデバッグ イベントが引き起こされた場合を考慮して下さい。仮にデバッグイベントが直ちに取得されたとすると、イベントハンドラからの復帰時に LDM が再実行されます。この場合、メモリの最初のワードに対して新しいアクセスが発生し、デバイスまたはストロングリオーダ メモリでは命令の各メモリ操作は正確に 1 回だけ発行されなければならないという規則が破られることになります。

C3.2.3 BKPT 命令デバッグイベント

BKPT 命令デバッグイベントは、BKPT 命令が実行にコミットされたときに発生します。BKPT は無条件命令です。

BKPT 命令デバッグイベントは同期です。つまり、デバッグイベントは、BKPT 命令をキャンセルする例外と同じように動作します。

BKPT 命令の詳細と、ARM および Thumb 命令セットでのエンコードについては、*BKPT* P. A8-56 を参照して下さい。

C3.2.4 ベクタキャッチデバッグ イベント

ベクタキャッチ レジスタ (DBGVCR) は、ベクタキャッチデバッグ イベントを制御します。*ベクタキャッチ レジスタ (DBGVCR)* P. C10-67 を参照して下さい。

ベクタキャッチデバッグ イベントは、次の場合に発生します。

- 命令の IVA が、現在のセキュリティ状態でのベクタアドレスと一致した。

  IVA の定義については、*メモリアドレス* P. C3-23 を参照して下さい。
- プロセッサがベクタキャッチの可能性をテストしたとき、DBGVCR の対応ビットが 1 にセットされていた。これは、ベクタキャッチが可能であることを示します。
- 命令が実行にコミットされている。デバッグイベントは、命令の条件コードチェックがパスしたか失敗したかにかかわらず生成されます。

ベクタキャッチデバッグ イベントのすべての条件が満たされると、プロセッサは実行中のモードにかかわらずイベントを生成します。

プロセッサは、命令を実行する前に、ベクタキャッチデバッグ イベントの可能性をテストする必要があります。

セキュリティ拡張機能が実装されていない場合、デバッグロジックは 1 組のベクタアドレスのみを使用して、ベクタキャッチデバッグ イベントを生成します。このアドレスの組を *ローカル ベクタ アドレス*と呼びます。

セキュリティ拡張機能が実装されている場合、デバッグロジックは次に示す 3 組のベクタアドレスを使用して、ベクタキャッチデバッグ イベントを生成します。

- 非セキュア例外モードで取得される例外用の組。これらは、*非セキュアローカルベクタ アドレス*と呼ばれます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- One set for exceptions taken in the Secure exception modes other than Monitor mode. These are called the *Secure Local vector addresses*.
- One set for exceptions taken in Monitor mode. These are called the *Monitor vector addresses*.

You enable vector catch independently for each of these vector addresses, by setting a bit in the DBGVCR to 1, see *Vector Catch Register (DBGVCR)* on page C10-67.

If the Security Extensions are not implemented, the debug logic determines whether to generate a Vector Catch debug event by comparing every instruction fetch with the Local vector addresses.

If the Security Extensions are implemented, the debug logic determines whether to generate a Vector Catch debug event by comparing every Secure instruction fetch with the Secure Local and Monitor vector addresses, and by comparing every Non-secure instruction fetch with the Non-secure Local vector addresses.

Note

Any instruction fetched from an exception vector address and committed for execution triggers a Vector Catch debug event if the appropriate bit in the DBGVCR is set to 1. Testing for possible Vector Catch debug events does not check whether the instruction is executed as a result of an exception entry.

Whether a Vector Catch debug event is generated for an instruction is UNPREDICTABLE if either:

- The exception vector address is word-aligned and one of the following applies:
  - — the first unit of the instruction is in the word at the exception vector address but is not at the exception vector address
  - — the first unit of the instruction is not in the word at the exception vector address but another unit of the instruction is in that word.

  This can occur when the processor is executing a variable-length instruction set, that is, in Thumb, ThumbEE or Jazelle state.
- The exception vector address is not word-aligned but is halfword-aligned and one of the following applies:
  - — The first unit of the instruction is in the halfword at the exception vector address but is not at the exception vector address. This can occur only in Jazelle state, where instructions consist of one or more byte-sized units.
  - — The first unit of the instruction includes the halfword at the exception vector address but is not at the exception vector address. This can occur only in ARM state, where all instructions are a single word and are word-aligned.
  - — The first unit of the instruction is not in the halfword at the exception vector address but another unit of the instruction is in that halfword. This can occur in variable-length instruction set states, that is, in Thumb, ThumbEE or Jazelle state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- モニタモード以外のセキュア例外モードで取得される例外用の組。これらは、*セキュア ローカルベクタ アドレス*と呼ばれます。
- モニタモードで取得される例外用の組。これらは、*モニタベクタ アドレス*と呼ばれます。

ベクタキャッチは、DBGVCR のビットを 1 にセットすることで、これらの各ベクタアドレスについて独立に可能にします。*ベクタキャッチ レジスタ (DBGVCR)* P. C10- 67 を参照して下さい。

セキュリティ拡張機能が実装されていない場合、デバッグロジックはすべての命令フェッチをローカルベクタ アドレスと比較し、ベクタキャッチデバッグ イベントを生成するかどうかを判定します。

セキュリティ拡張機能が実装されている場合、デバッグロジックはすべてのセキュア命令フェッチをセキュアローカルおよびモニタベクタ アドレスと比較し、すべての非セキュア命令フェッチを非セキュアローカルベクタ アドレスと比較することで、ベクタキャッチ デバッグイベントを生成するかどうかを判定します。

注

例外ベクタアドレスからフェッチされ、実行にコミットされた命令は、DBGVCR の対応するビットが 1 にセットされている場合に、ベクタキャッチデバッグ イベントをトリガします。ベクタキャッチデバッグ イベントの可能性に対するテストでは、例外開始の結果として命令が実行されるかどうかのチェックは行われません。

次のいずれかの場合、命令についてベクタキャッチデバッグ イベントが生成されるかどうかは予測不能です。

- 例外ベクタアドレスがワードアラインドで、次のいずれかが適用される
  - – 命令の最初の単位が例外ベクタアドレスのワードに含まれているが、例外ベクタアドレスには存在しない。
  - – 命令の最初の単位が、例外ベクタアドレスのワードに含まれていないが、命令の別の単位がそのワードに含まれている。

  この状況は、プロセッサが可変長命令セットを実行している場合、つまり Thumb、ThumbEE、Jazelle のいずれかの状態で発生する可能性があります。
- 例外ベクタアドレスがワードアラインドではなくハーフワードアラインドで、次のいずれかが適用される。
  - – 命令の最初の単位が、例外ベクタアドレスのハーフワードに含まれているが、例外ベクタアドレスには存在しない。この状況は、命令が 1 つ以上のバイト単位で構成される Jazelle 状態でのみ発生する可能性があります。
  - – 命令の最初の単位に、例外ベクタアドレスのハーフワードが含まれているが、その単位が例外ベクタアドレスには存在しない。この状況は、すべての命令が 1 ワードかつワードアラインドである ARM 状態でのみ発生する可能性があります。
  - – 命令の最初の単位が例外ベクタアドレスのハーフワードに含まれていないが、命令の別の単位がそのハーフワードに含まれている。この状況は、可変長命令セットの状態である Thumb、ThumbEE、Jazelle 状態で発生する可能性があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

Normally, exception vector addresses must be word-aligned. However, when SCTLR.VE == 1, enabling vectored interrupt support, the exception vector address for one or both of the IRQ and FIQ vectors might not be word-aligned. Support for exception vector addresses that are not word-aligned is IMPLEMENTATION DEFINED, see *Vectored interrupt support* on page B1-32.

If Monitor debug-mode is selected and enabled, and the vector is either the Prefetch Abort vector or the Data Abort vector, the debug event is:

- UNPREDICTABLE in v7 Debug
- ignored in v6 Debug and v6.1 Debug.

Vector Catch debug events are synchronous. That is, the debug event acts like an exception that cancels the instruction at the caught vector. When invasive debug is enabled and Monitor debug-mode is selected, if Vector Catch debug events are permitted a Vector Catch debug event generates a Prefetch Abort exception. For more information, see *Generation of debug events* on page C3-40.

Note

A Vector Catch debug event is taken only when the instruction is committed for execution and therefore might not be taken if another exception occurs, see *Debug event prioritization* on page C3-43.

For more information, see *Vector Catch Register (DBGVCR)* on page C10-67.

Vector catch debug events and vectored interrupt support

The ARM architecture provides support for vectored interrupts, where an interrupt controller provides the interrupt vector address directly to the processor. The mechanism for defining the vectors is IMPLEMENTATION DEFINED. You enable the use of vectored interrupts by setting the SCTLR.VE bit to 1. For more information see *Vectored interrupt support* on page B1-32.

Vectored interrupt support affects Vector Catch debug event generation for the IRQ and FIQ exception vectors. These two vectors are described as the *interrupt vectors*. The details of Vector Catch debug event generation on the interrupt vectors depend on whether the Security Extensions are implemented:

If the Security Extensions are not implemented

- If the SCTRL.VE bit is set to 0, then the Local vector addresses for IRQ and FIQ vector catch are determined by the exception base address.
- If the SCTRL.VE bit is set to 1, then the Local vector address for an IRQ or FIQ vector catch is the interrupt vector address supplied by the interrupt controller on taking the interrupt.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

通常は、例外ベクタアドレスはワードアラインドの必要があります。ただし、SCTLR.VE = 1 にしてベクタ割り込みサポートを可能にすると、IRQ および FIQ ベクタの 1 つまたは両方の例外ベクタアドレスは、ワードアラインドでない場合があります。ワードアラインドでない例外ベクタアドレスのサポートは実装定義です。*ベクタ割り込みのサポート* P. B1-32 を参照して下さい。

モニタデバッグ モードが選択され可能であり、ベクタがプリフェッチアボート ベクタまたはデータアボート ベクタの場合、デバッグイベントは次のように動作します。

- v7 デバッグでは予測不能です。
- v6 デバッグおよび v6.1 デバッグでは無視されます。

ベクタキャッチデバッグ イベントは同期です。つまり、このデバッグイベントは、キャッチされたベクタにある命令をキャンセルする例外と同じように動作します。侵襲性デバッグが可能で、モニタデバッグ モードが選択されている場合、ベクタキャッチデバッグ イベントが許可されていれば、ベクタキャッチデバッグ イベントはプリフェッチアボート例外を生成します。詳細については、*デバッグ イベントの生成* P. C3-40 を参照して下さい。

注

ベクタキャッチデバッグ イベントは、命令が実行にコミットされたときのみ取得されます。そのため、別の例外が発生すると、ベクタキャッチデバッグ イベントは取得されないことがあります。*デバッグイベントの優先度付け* P. C3-43 を参照して下さい。

詳細については、*ベクタキャッチ レジスタ (DBGVCR)* P. C10-67 を参照して下さい。

ベクタキャッチデバッグ イベントとベクタ割り込みのサポート

ARM アーキテクチャはベクタ割り込みのサポートを提供しており、割り込みコントローラが割り込みベクタアドレスを直接プロセッサへ提供します。ベクタを定義する機構は実装定義です。ベクタ割り込みの使用を可能にするには、SCTLR.VE ビットを 1 にセットします。詳細については、*ベクタ割り込みのサポート* P. B1-32 を参照して下さい。

ベクタ割り込みのサポートは、IRQ および FIQ 例外ベクタのベクタキャッチ デバッグイベントの生成に影響します。これら 2 つのベクタを、*割り込みベクタ*と呼びます。割り込みベクタへのベクタキャッチデバッグ イベント生成の詳細は、セキュリティ拡張機能が実装されているかどうかに依存します。

セキュリティ拡張機能が実装されていない場合

- SCTRL.VE ビットが 0 にセットされている場合、IRQ および FIQ ベクタ キャッチのローカルベクタ アドレスは例外のベースアドレスにより決定されます。
- SCTRL.VE ビットが 1 にセットされている場合、IRQ または FIQ ベクタ キャッチのローカルベクタ アドレスは、割り込み取得時に割り込みコントローラから供給される割り込みベクタアドレスです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If the Security Extensions are implemented

The Secure Local and Non-secure Local vector addresses for IRQ and FIQ vector catch are determined by the appropriate banked copy of the SCTRL.VE bit:

- If the SCTRL.VE bit is set to 0, then the corresponding Local vector addresses for IRQ and FIQ vector catch are determined by the banked exception base address.
- If the SCTRL.VE bit is set to 1, then for each of IRQ and FIQ vector catch:
  - — if the interrupt is taken in Secure or Non-Secure IRQ mode or FIQ mode, then the corresponding Local vector address is the interrupt vector address supplied by the interrupt controller on taking the interrupt.
  - — if the interrupt is taken in Monitor mode, then it is IMPLEMENTATION DEFINED whether the IRQ and FIQ Vector Catch debug events generated from the Local vector addresses can occur, and if they can occur the Secure and Non-secure Local vector addresses for the vector catches are IMPLEMENTATION DEFINED.

The Monitor vector addresses for IRQ and FIQ vector catch are determined by the Monitor exception base address.

When the Vector Catch debug logic uses addresses supplied by the interrupt controller, then:

- if the interrupt controller has not supplied an interrupt address to the processor since vectored interrupt support was enabled then no Vector Catch debug events using Local vector addresses are generated
- if Vector Catch debug events were not enabled when the interrupt controller supplied a vector address to the processor, but have been enabled since, an implementation must consistently either:
  - — generate a Vector Catch debug event if the IVA of an instruction matches the Local vector address
  - — not generate Vector Catch debug events using any Local vector address.

C3.2.5 Memory addresses

On processors that implement the *Virtual Memory System Architecture* (VMSA), and also implement the *Fast Context Switch Extension* (FCSE):

- It is IMPLEMENTATION DEFINED whether the *Instruction Virtual Address* (IVA) used in generating Breakpoint debug events is the *Modified Virtual Address* (MVA) or *Virtual Address* (VA) of the instruction.
- It is IMPLEMENTATION DEFINED whether the *Data Virtual Address* (DVA) used in generating Watchpoint debug events is the MVA or VA of the data access.
- The IVA used in generating Vector Catch debug events is always the VA of the instruction.
- The *Watchpoint Fault Address Register* (DBGWFAR) reads a VA plus an offset that depends on the processor instruction set state.
- The *Program Counter Sampling Register* (DBGPCSR), if implemented, reads a VA plus an offset that depends on the processor instruction set state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

セキュリティ拡張機能が実装されている場合

IRQ および FIQ ベクタキャッチのセキュアローカルおよび非セキュアローカルベクタ アドレスは、SCTRL.VE ビットの対応するバンクのコピーにより決定されます。

- SCTRL.VE ビットが 0 にセットされている場合、IRQ および FIQ ベクタキャッチの対応するローカルベクタ アドレスは、バンクされた例外ベースアドレスにより決定されます。
- SCTRL.VE ビットが 1 にセットされている場合、各 IRQ および FIQ ベクタキャッチについて、次のように決定されます。
  - – セキュアまたは非セキュアの IRQ モードまたは FIQ モードで割り込みが取得された場合、対応するローカルベクタ アドレスは、割り込みの取得時に割り込みコントローラから供給される割り込みベクタ アドレスです。
  - – モニタモードで割り込みが取得された場合、ローカルベクタ アドレスから生成される IRQ および FIQ ベクタキャッチデバッグ イベントが発生するかどうかは実装定義で、発生する場合には、ベクタキャッチのセキュアおよび非セキュアローカルベクタ アドレスは実装定義です。

IRQ および FIQ ベクタキャッチのモニタベクタ アドレスは、モニタ例外ベースアドレスによって決定されます。

ベクタキャッチデバッグ ロジックが、割り込みコントローラによって供給されるアドレスを使用する場合、次の条件が適用されます。

- ベクタ割り込みサポートが可能になった後で、割り込みコントローラがプロセッサに割り込みアドレスを供給していない場合、ローカルベクタ アドレスを使用するベクタキャッチデバッグ イベントは生成されません。
- 割り込みコントローラがプロセッサにベクタアドレスを供給したときにベクタキャッチデバッグ イベントが可能でなかったが、その後で可能になった場合、実装は一貫して次のいずれかの動作を行う必要があります。
  - – 命令の IVA がローカルベクタ アドレスと一致していれば、ベクタキャッチデバッグ イベントを生成する。
  - – どのローカルベクタ アドレスを使用するベクタキャッチデバッグ イベントも生成しない。

C3.2.5 メモリアドレス

仮想メモリシステム アーキテクチャ(VMSA) と*高速コンテキストスイッチ拡張機能*(FCSE) の両方を実装しているプロセッサでは、次の条件が適用されます。

- ブレークポイントデバッグ イベントの生成に使用される*命令仮想アドレス*(IVA) が、命令の*修飾仮想アドレス*(MVA) と*仮想アドレス*(VA) のどちらであるかは実装定義です。
- ウォッチポイントデバッグ イベントの生成に使用される*データ仮想アドレス*(DVA) が、データアクセスの MVA と VA のどちらであるかは実装定義です。
- ベクタキャッチ デバッグイベントの生成に使用される IVA は、常に命令の VA です。
- *ウォッチポイントフォールトアドレス レジスタ*(DBGWFAR) の読み出し値は、VA に、プロセッサの命令セット状態によって異なるオフセットを加えた値です。
- *プログラムカウンタ サンプリングレジスタ*(DBGPCSR) が実装されている場合の読み出し値は、VA に、プロセッサの命令セット状態によって異なるオフセットを加えた値です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

The FCSE is optional in ARMv7, and ARM deprecates use of the FCSE.

On processors that implement the VMSA, and do not implement the FCSE:

- The IVA used in generating Breakpoint debug events is the VA of the instruction.
- The DVA used in generating Watchpoint debug events is the VA of the data access.
- The IVA used in generating Vector Catch debug events is the VA of the instruction.
- The DBGWFAR reads a VA plus an offset that depends on the processor instruction set state.
- The DBGPCSR reads a VA plus an offset that depends on the processor instruction set state.

On processors that implement the *Protected Memory System Architecture* (PMSA), the Virtual Address is identical to the *Physical Address* (PA) and therefore:

- The IVA used in generating Breakpoint debug events is the PA of the instruction.
- The DVA used in generating Watchpoint debug events is the PA of the data access.
- The IVA used in generating Vector Catch debug events is the PA of the instruction.
- The DBGWFAR reads a PA plus an offset that depends on the processor instruction set state.
- The DBGPCSR reads a PA plus an offset that depends on the processor instruction set state.

For more information about the DBGWFAR, see:

- *Effects of debug exceptions on CP15 registers and the DBGWFAR* on page C4-4
- *Effect of entering Debug state on CP15 registers and the DBGWFAR* on page C5-4
- *Watchpoint Fault Address Register (DBGWFAR)* on page C10-28.

For more information about the DBGPCSR, see *Program Counter sampling* on page C8-2 and *Program Counter Sampling Register (DBGPCSR)* on page C10-38.

C3.2.6 UNPREDICTABLE behavior on Software debug events

In ARMv6 the following events are ignored if Monitor debug-mode is configured, because they could lead to an unrecoverable state:

- Vector Catch debug events on the Prefetch Abort and Data Abort vectors
- Unlinked Context ID Breakpoint debug events, if the processor is running in a privileged mode
- Linked or Unlinked Instruction Virtual Address mismatch Breakpoint debug events, if the processor is running in a privileged mode.

In ARMv7, if Monitor debug-mode is configured the generation of the following events is UNPREDICTABLE and can lead to an unrecoverable state:

- Vector Catch debug events on the Prefetch Abort and Data Abort vectors
- Unlinked Context ID Breakpoint debug events that are configured to be generated in any mode, or to be generated only in privileged modes
- Linked or Unlinked Instruction Virtual Address mismatch Breakpoint debug events that are configured to be generated in any mode, or to be generated only in privileged modes.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

FCSE は ARMv7 ではオプションで、使用は推奨されません。

VMSA を実装し、FCSE を実装していないプロセッサでは、次の条件が適用されます。

- ブレークポイントデバッグ イベントの生成に使用される IVA は、命令の VA です。
- ウォッチポイントデバッグ イベントの生成に使用される DVA は、データアクセスの VA です。
- ベクタキャッチ デバッグイベントの生成に使用される IVA は、命令の VA です。
- DBGWFAR の読み出し値は、VA に、プロセッサの命令セット状態によって異なるオフセットを加えた値です。
- DBGPCSR の読み出し値は、VA に、プロセッサの命令セット状態によって異なるオフセットを加えた値です。

保護メモリシステム アーキテクチャ(PMSA) を実装しているプロセッサでは、仮想アドレスは*物理アドレス*(PA) と同じなので、次の条件が適用されます。

- ブレークポイントデバッグ イベントの生成に使用される IVA は、命令の PA です。
- ウォッチポイントデバッグ イベントの生成に使用される DVA は、データアクセスの PA です。
- ベクタキャッチデバッグ イベントの生成に使用される IVA は、命令の PA です。
- DBGWFAR の読み出し値は、PA に、プロセッサの命令セット状態によって異なるオフセットを加えた値です。
- DBGPCSR の読み出し値は、PA に、プロセッサの命令セット状態によって異なるオフセットを加えた値です。

DBGWFAR の詳細については、次に示すセクションを参照して下さい。

- *CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響* P. C4- 4
- *デバッグ状態への遷移が CP15 レジスタおよび DBGWFAR に及ぼす影響* P. C5- 4
- *ウォッチポイント フォールトアドレス レジスタ (DBGWFAR)* P. C10- 28

DBGPCSR の詳細については、*プログラムカウンタのサンプリング* P. C8- 2 および*プログラムカウンタサンプリング レジスタ (DBGPCSR)* P. C10- 38 を参照して下さい。

C3.2.6 ソフトウェアデバッグ イベントでの予測不能な動作

ARMv6 では、モニタデバッグ モードが構成されている場合、次に示すイベントは回復不能な状態を引き起こす可能性があるために無視されます。

- プリフェッチアボートおよびデータアボート ベクタでのベクタキャッチデバッグ イベント
- プロセッサが特権モードで実行されている場合の、リンクなしのコンテキスト ID ブレークポイントデバッグ イベント
- プロセッサが特権モードで実行されている場合の、リンク付きまたはリンクなしの命令仮想アドレス不一致ブレークポイントデバッグ イベント

ARMv7 では、モニタデバッグ モードが構成されている場合には、次に示すイベントの生成は予測不能で、回復不能な状態を引き起こす可能性があります。

- プリフェッチアボートおよびデータアボート ベクタでのベクタキャッチデバッグ イベント
- すべてのモード、または特権モードでのみ生成されるよう構成されている、リンクなしのコンテキスト ID ブレークポイントデバッグ イベント
- すべてのモード、または特権モードでのみ生成されるよう構成されている、リンク付きまたはリンクなしの命令仮想アドレス不一致ブレークポイントデバッグ イベント

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When Monitor debug-mode is configured, debuggers must avoid these cases by restricting the programming of the debug event control registers:

- DBGVCR[28,27,12,11,4,3] must be programmed as zero, see *Vector Catch Register (DBGVCR)* on page C10-67.
- The permitted values of the Privileged Mode control bits, DBGBCR[2:1], must be restricted in the following cases:
  - — if DBGBCR[22:20] is set to 0b010, selecting an Unlinked Context ID breakpoint
  - — If DBGBCR[22:20] is set to 0b100 or 0b101, selecting an IVA mismatch breakpoint.

  For these cases, DBGBCR[2:1] must be programmed to one of:
  - — 0b00, selecting match only in User, Supervisor or System mode
  - — 0b10, selecting match only in User mode.

  See *Debug exceptions in abort handlers* for additional points that must be considered before using the 0b00 setting.

  For details of programming the DBGBCR see *Breakpoint Control Registers (DBGBCR)* on page C10-49.

If these restrictions are not followed, processor behavior on a resulting debug event is UNPREDICTABLE.

When the Security Extensions are implemented Vector Catch debug events on the Secure Monitor Call vector are not ignored and are not UNPREDICTABLE. However, normally DBGVCR[10] is also programmed as zero, see *Monitor debug-mode vector catch on Secure Monitor Call* on page C3-26.

Debug exceptions in abort handlers

The previous section indicated that, in ARMv7, a debugger might set DBGBCR[2:1] to 0b00, match in User, Supervisor and System modes, to avoid the possibility of reaching an unrecoverable state in the Unlinked Context ID and IVA mismatch breakpoint cases when Monitor debug-mode is selected. However, DBGBCR[2:1] must only be programmed to 0b00 if you are confident that the abort handler will not switch to one of these modes before saving context that might be corrupted by an additional debug event. The context that might be corrupted by such an event includes LR_abt, SPSR_abt, IFAR, DFAR, and DFSR.

It is unlikely that an abort handler would switch to User mode to process an abort before saving these registers, so setting DBGBCR[2:1] to 0b10, match only in User mode, is safer.

Also, take care when setting a Breakpoint or BKPT Instruction debug event inside a Prefetch Abort or Data Abort handler, or when setting a Watchpoint debug event on a data address that might be accessed by any of these handlers.

In general, a user must only set Breakpoint or BKPT Instruction debug events inside an abort handler at a point after the context that would be corrupted by a debug event has been saved. Breakpoint debug events in code that might be run by an abort handler can be avoided by setting DBGBCR[2:1] to 0b00 or 0b01, as appropriate.

Watchpoint debug events in abort handlers can be avoided by setting DBGWCR[2:1] for the watchpoint to 0b10, match only unprivileged accesses, if the code being debugged is not running in a privileged mode.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

モニタデバッグ モードが構成されているとき、デバッガはデバッグイベント制御レジスタのプログラミングを次のように制限し、これらの状態を回避する必要があります。

- DBGVCR[28、27、12、11、4、3] は 0 としてプログラムする必要があります。*ベクタキャッチ レジスタ (DBGVCR)* P. C10- 67 を参照して下さい。
- 次の条件では、特権モード制御ビット (DBGBCR[2:1]) で許可される値を制限する必要があります。
  - – DBGBCR[22:20] が 0b010 に設定され、リンクなしコンテキスト ID ブレークポイントが選択されている場合
  - – DBGBCR[22:20] が 0b100 または 0b101 に設定され、IVA 不一致ブレークポイントが選択されている場合

  これらの条件では、DBGBCR[2:1] を次のいずれかにプログラムする必要があります。
  - – 0b00：ユーザ、スーパバイザ、システムモードのみでの一致を選択します。
  - – 0b10：ユーザモードでの一致のみを選択します。

  0b00 設定を使用する前に考慮する必要がある追加事項については、*アボートハンドラでのデバッグ例外*を参照して下さい。

  DBGBCR のプログラミングの詳細については、*ブレークポイント制御レジスタ (DBGBCR)* P. C10- 49 を参照して下さい。

これらの制限に従わない場合、結果としてデバッグイベントが発生したときのプロセッサの動作は予測不能です。

セキュリティ拡張機能が実装されている場合、セキュアモニタコール ベクタでのベクタキャッチデバッグ イベントは無視されず、予測不能でもありません。ただし、通常は DBGVCR[10] も 0 としてプログラムされます。*セキュアモニタ コールでのモニタデバッグ モードのベクタキャッチ* P. C3- 26 を参照して下さい。

アボートハンドラでのデバッグ例外

前のセクションで説明したように、ARMv7 でのデバッガは、モニタデバッグ モードが選択されているとき、リンクなしコンテキスト ID および IVA 不一致ブレークポイントの状況で回復不能な状態に到達する可能性を回避するため、DBGBCR[2:1] を 0b00 に設定し、ユーザ、スーパバイザ、システムモードのみでの一致を選択することができます。ただし、DBGBCR[2:1] を 0b00 にプログラムするのは、追加のデバッグイベントにより破壊される可能性があるコンテキストを保存する前に、アボートハンドラがこれらのいずれかのモードに切り替わる可能性がないことが確実な場合のみ行って下さい。このようなイベントで破壊される可能性があるコンテキストには、LR_abt、SPSR_abt、IFAR、DFAR、DFSR があります。

アボートハンドラが、これらのレジスタを保存する前に、アボート処理のためユーザモードに切り替わることは考えにくいため、DBGBCR[2:1] を 0b10 に設定し、ユーザモードでの一致のみを選択するのがより安全です。

また、プリフェッチアボートまたはデータアボート ハンドラ内に、ブレークポイントまたは BKPT 命令デバッグイベントを設定する場合、あるいはこれらのハンドラのいずれかによってアクセスされる可能性があるデータアドレスにウォッチポイントデバッグ イベントを設定する場合は注意が必要です。

一般に、ユーザがアボートハンドラ内にブレークポイントまたは BKPT 命令デバッグイベントを設定するのは、デバックイベントにより破壊されるコンテキストが保存された後でのみ行う必要があります。アボートハンドラによって実行される可能性があるコード内のブレークポイントデバッグイベントは、DBGBCR[2:1] を 0b00 または 0b01 の適切な値に設定することで回避できます。

アボートハンドラ内のウォッチポイントデバッグ イベントは、デバッグ対象のコードが特権モードで実行されていなければ、ウォッチポイントの DBGWCR[2:1] を 0b10 に設定し、非特権アクセスにのみ一致するようにすることで回避できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If these guidelines are not followed, a debug event might occur before the handler has saved the context of the abort, causing the context to be overwritten. This loss of context results in UNPREDICTABLE software behavior. The context that might be corrupted by such an event includes LR_abt, SPSR_abt, IFAR, DFAR, and DFSR.

Debug events in the debug monitor

Because debug exceptions generate Data Abort or Prefetch Abort exceptions, the precautions outlined in the section *Debug exceptions in abort handlers* on page C3-25 also apply to debug monitors. The suggested settings for breakpoints and watchpoints that can avoid taking debug exceptions in a Data Abort handler can be used to avoid taking debug exceptions in the debug monitor.

In addition, particularly on ARMv7 processors that do not implement the Extended CP14 interface, and particularly those that implement synchronous Watchpoint debug events, when Monitor debug-mode is enabled debuggers must avoid:

- setting Watchpoint debug events on the addresses of debug registers
- setting Breakpoint and Vector Catch debug events on the addresses of instructions in the debug monitor.

In particular, it is unwise to set a watchpoint on the address of the Watchpoint Control Register (DBGWCR) for that watchpoint, or to set a breakpoint on the address of an instruction that disables the breakpoint.

The section *Generation of debug events* on page C3-40 identifies two problem cases:

- A write to the DBGWCR for a watchpoint set on the address of that DBGWCR, to disable that watchpoint, triggers the watchpoint.

  In this case:

  — if watchpoints are asynchronous, the write to the DBGWCR still takes place and the watchpoint is disabled. The debug software must then deal with the re-entrant debug exception.

  — if watchpoints are synchronous the value in the DBGWCR after the watchpoint is signaled is unchanged, and the debug event is left enabled.

- an instruction that disables a breakpoint on that instruction triggers the breakpoint.

  In this case, the debug exception is taken before the debug event is disabled.

In both of these cases it might be impossible to recover.

Monitor debug-mode vector catch on Secure Monitor Call

Debuggers must be cautious about programming a Vector Catch debug event on the Secure Monitor Call (SMC) vector when Monitor debug-mode is configured. If such an event is programmed, the following sequence can occur:

1. Non-secure code executes an `SMC` instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

これらのガイドラインに従わない場合、ハンドラがアボートのコンテキストを保存する前にデバッグイベントが発生し、コンテキストが上書きされる可能性があります。コンテキストが消失した場合、ソフトウェアの動作は予測不能です。このようなイベントで破壊される可能性があるコンテキストには、LR_abt、SPSR_abt、IFAR、DFAR、DFSR があります。

デバッグモニタでのデバッグイベント

デバッグ例外はデータアボートまたはプリフェッチアボート例外を生成するため、*アボートハンドラでのデバッグ例外* P. C3-25 で概説した注意事項はデバッグモニタにも適用されます。データアボート ハンドラでデバッグ例外が取得されることを回避するために推奨されるブレークポイントおよびウォッチポイントの設定は、デバッグモニタでデバッグ例外が取得されることを回避するためにも使用できます。

さらに、拡張 CP14 インタフェースを実装していない ARMv7 プロセッサでは特に、また同期ウォッチポイントデバッグ イベントを実装しているプロセッサでは特に、モニタデバッグモードが可能なときは、デバッガは次の動作を避ける必要があります。

- デバッグレジスタのアドレスにウォッチポイントデバッグ イベントを設定する。
- デバッグモニタの命令のアドレスに、ブレークポイントおよびベクタキャッチデバッグ イベントを設定する。

特に、ウォッチポイントを、そのウォッチポイントのウオッチポイント制御レジスタ (DBGWCR) のアドレスに設定する、またはブレークポイントを、そのブレークポイントを非稼動にする命令のアドレスに設定することは好ましくありません。

デバッグ イベントの生成 P. C3-40 には、問題となる次の 2 つの状況が示されています。

- ウォッチポイントが、自分自身の DBGWCR のアドレスに設定されており、そのウォッチポイントを非稼動にするために DBGWCR へ書き込みを行う動作が、そのウォッチポイントをトリガする場合。

  この場合、次の動作が行われます。

  – ウォッチポイントが非同期の場合、DBGWCR への書き込みが行われ、ウォッチポイントは非稼動になります。その後で、デバッグソフトウェアはリエントラントなデバッグ例外を処理する必要があります。

  – ウォッチポイントが同期の場合、ウォッチポイントが起動された後の DBGWCR の値は変更されず、デバッグイベントは可能なままです。

- その命令に対してのブレークポイントを非稼動にする命令が、そのブレークポイントをトリガする場合。

  この場合、デバッグイベントが不可能になる前に、デバッグ例外が取得されます。

これらの両方の状況で、回復が不可能になる可能性があります。

セキュアモニタ コールでのモニタデバッグ モードのベクタキャッチ

モニタデバッグ モードが構成されているとき、セキュアモニタ コール (SMC) ベクタに対するベクタキャッチデバッグ イベントをプログラムする場合、デバッグ開発者は注意が必要です。このようなイベントがプログラムされている場合、次のようなシーケンスが発生する可能性があります。

1. 非セキュアコードが、`SMC` 命令を実行します。
2. プロセッサが `SMC` 例外を取得し、モニタモードでモニタベクタへ分岐します。SCR.NS ビットは 1 にセットされ、`SMC` が非セキュア状態で実行されたことを示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

2. The processor takes the SMC exception, branching to the Monitor vector in Monitor mode. The SCR.NS bit is set to 1, indicating the SMC originated in the Non-secure state.
3. The Vector Catch debug event is taken. Although SCR.NS is set to 1, the processor is in the Secure state because it is in Monitor mode.
4. The processor jumps to the Secure Prefetch Abort vector, and sets SCR.NS to 0.

   Note

   Aborts taken in Secure state cause SCR.NS to be set to 0.

5. The abort handler at the Secure Prefetch Abort handler can tell a Vector Catch debug event occurred, and can determine the address of the SMC instruction from LR_mon. However, it cannot determine whether that is a Secure or Non-secure address.

Therefore, ARM recommends that you do not program a Vector Catch debug event on the SMC vector when Monitor debug-mode is enabled.

Note

This is not a security issue, because the sequence given here can only occur if **SPIDEN** is HIGH.

Possible effect of the Security Extensions on FIQ vector catch

When the Security Extensions are implemented, a debugger might need to consider the implications of the SCR on a Vector Catch event set on the FIQ vector, when the SCR is configured with both:

- the SCR.FW bit set to 0, so the CPSR.F bit cannot be modified in Non-secure state
- the SCR.FIQ bit set to 0, so that FIQs are handled in FIQ mode.

With this configuration, if an FIQ occurs in Non-secure state, the processor does not set CPSR.F to disable FIQs, and so the processor repeatedly takes the FIQ exception.

It might not be possible to debug this situation using the vector catch on FIQ because the instruction at the FIQ exception vector is never committed for execution and therefore the debug event never occurs.

C3.2.7 Pseudocode details of Software debug events

The following subsections give pseudocode details of Software debug events:

- *Debug events*
- *Breakpoints and Vector Catches* on page C3-28
- *Watchpoints* on page C3-35.

Debug events

The following functions cause the corresponding debug events to occur:

```
BKPTInstrDebugEvent()
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

3. ベクタキャッチデバッグ イベントが取得されます。SCR.NS は 1 にセットされていますが、プロセッサはモニタモードなのでセキュア状態です。
4. プロセッサはセキュアプリフェッチアボート ベクタにジャンプし、SCR.NS を 0 にセットします。

   注

   セキュア状態で取得されたアボートにより、SCR.NS は 0 にセットされます。

5. セキュアプリフェッチアボート ハンドラのアボートハンドラは、ベクタキャッチデバッグ イベントが発生したことを検出でき、SMC 命令のアドレスを LR_mon から特定できます。しかし、そのアドレスがセキュアまたは非セキュアのどちらであるかは判別できません。

この理由から、モニタデバッグ モードが可能な場合、SMC ベクタにベクタキャッチデバッグ イベントをプログラムしないことをお勧めします。

注

ここで示したシーケンスは SPIDEN が HIGH の場合のみ発生し得るため、セキュリティの問題ではありません。

FIQ ベクタキャッチがセキュリティ拡張機能に及ぼす可能性がある影響

セキュリティ拡張機能が実装されている場合、デバッガは、SCR が次の両方に構成されているときに、FIQ ベクタに設定されているベクタキャッチ イベントが SCR に与える影響を考慮することが必要な場合があります。

- SCR.FW ビットが 0 にセットされ、非セキュア状態での CPSR.F ビットの変更が禁止されている。
- SCR.FIQ ビットが 0 にセットされ、FIQ が FIQ モードで処理されるよう設定されている。

この構成で、FIQ が非セキュア状態で発生すると、プロセッサは CPSR.F をセットして FIQ を不可能にする動作を行わないため、プロセッサが FIQ 例外を繰り返し取得することになります。

この場合、FIQ 例外ベクタにある命令は実行にコミットされず、デバッグイベントが発生しないため、FIQ 上のベクタキャッチを使用してこの状況をデバッグすることができない可能性があります。

C3.2.7 ソフトウェアデバッグ イベントの擬似コードの詳細

次に示すサブセクションでは、ソフトウェアデバッグ イベントの擬似コードの詳細について解説しています。

- *デバッグイベント*
- *ブレークポイントとベクタキャッチ* P. C3-28
- *ウォッチポイント* P. C3-35

デバッグイベント

次に示す関数では、対応するデバッグイベントが発生します。

```
BKPTInstrDebugEvent()
BreakpointDebugEvent()
VectorCatchDebugEvent()
WatchpointDebugEvent()
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
BreakpointDebugEvent()
VectorCatchDebugEvent()
WatchpointDebugEvent()
```

If the debug event is not permitted, it is ignored by the processor.

Breakpoints and Vector Catches

If invasive debug is enabled, on each instruction the `Debug_CheckInstruction()` function checks for BRP and DBGVCR matches. If a match is found the function calls `BreakpointDebugEvent()` or `VectorCatchDebugEvent()`. If the debug event is not permitted, it is ignored by the processor.

On a simple sequential execution model, the `Debug_CheckInstruction()` call for an instruction occurs just before the Operation pseudocode for the instruction is executed, and any call it generates to `BreakpointDebugEvent()` or `VectorCatchDebugEvent()` must happen at that time. However, the architecture does not define when the checks for BRP and DBGVCR matches are made, other than that they must be made at or before that time. Therefore an implementation can perform the checks much earlier in an instruction pipeline, marking the instruction as breakpointed, and cause a marked instruction to call `BreakpointDebugEvent()` or `VectorCatchDebugEvent()` if and when it is about to execute.

The `BRPMatch()` function checks an individual BRP match, calling the `BRPLinkMatch()` function if necessary to check whether a linked BRP matches.

The `VCRMatch()` function checks for a Vector Catch debug event. When vectored interrupt support is enabled, it uses variables to hold the IRQ and FIQ interrupt vector addresses supplied to the processor by the interrupt controller on taking an interrupt in IRQ mode or FIQ mode. These variables are updated by the `VCR_OnTakingInterrupt()` function, that is called each time the processor takes an IRQ or FIQ interrupt.

For all of these functions, between a context changing operation and an exception entry, exception return or explicit *Instruction Synchronization Barrier* (ISB) operation, it is UNPREDICTABLE whether the values of `CurrentModeIsPrivileged()`, CPSR.M, `CurrentInstrSet()`, `FindSecure()`, and the CONTEXTIDR used by `BRPMatch()`, `BRPLinkMatch()`, and `VCRMatch()` are the old or the new values.

```
// Debug_CheckInstruction()
// ========================

Debug_CheckInstruction(bits(32) address, integer length)

    // Do nothing if debug disabled.
    if DBGDSCR<15:14> == '00' then return;

    case CurrentInstrSet() of
        when InstrSet_ARM
            step = 4;
        when InstrSet_Thumb, InstrSet_ThumbEE
            step = 2;
        when InstrSet_Jazelle
            step = 1;
    length = length / step;

    vcr_match = FALSE;
    brp_match = FALSE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

デバッグイベントが許可されていない場合、プロセッサにより無視されます。

ブレークポイントとベクタキャッチ

侵襲性デバッグが可能な場合、各命令について `Debug_CheckInstruction()` 関数により BRP および DBGVCR 一致がチェックされます。一致が見つかった場合、この関数により `BreakpointDebugEvent()` または `VectorCatchDebugEvent()` が呼び出されます。デバッグイベントが許可されていない場合、プロセッサにより無視されます。

単純なシーケンシャルの例外モデルでは、命令についての `Debug_CheckInstruction()` 呼び出しは、命令の動作擬似コードが実行される直前に発生し、それによって `BreakpointDebugEvent()` または `VectorCatchDebugEvent()` への呼び出しが生成される場合、その呼び出しもこの時点で発生する必要があります。ただし、BRP および DBGVCR 一致のチェックがどの時点で行われるかについて、この時点またはそれ以前に行われる必要があるという以外は、アーキテクチャにより定義されていません。このため、実装では、命令パイプラインのはるかに早期の段階でチェックを実行し、命令をブレークポイント対象としてマークし、その命令が実行段階に達した場合、その時点で `BreakpointDebugEvent()` または `VectorCatchDebugEvent()` への呼び出しを、マークされた命令について発生させるという動作を行うことができます。

`BRPMatch()` 関数は個別の BRP 一致をチェックし、リンク付き BRP が一致するかどうかのチェックを行うために必要な場合は `BRPLinkMatch()` 関数を呼び出します。

`VCRMatch()` 関数は、ベクタキャッチデバッグ イベントをチェックします。ベクタ割り込みサポートが可能な場合、この関数は、IRQ モードまたは FIQ モードでの割り込み取得時に、割り込みコントローラからプロセッサへ供給される IRQ および FIQ 割り込みベクタアドレスを保持するため、変数を使用します。これらの変数は、プロセッサが IRQ または FIQ 割り込みを取得するごとに呼び出される `VCR_OnTakingInterrupt()` 関数によって更新されます。

これらの関数すべてについて、コンテキスト変更操作と、例外開始、例外からの復帰、または明示的な*命令同期バリア*(ISB) 操作との間で、`CurrentModeIsPrivileged()`、CPSR.M、`CurrentInstrSet()`、`FindSecure()`、および `BRPMatch()`、`BRPLinkMatch()`、`VCRMatch()` で使用される CONTEXTIDR が、古い値と新しい値のどちらであるかは予測不能です。


```
// Debug_CheckInstruction()
// ========================

Debug_CheckInstruction(bits(32) address, integer length)

    // Do nothing if debug disabled.
    if DBGDSCR<15:14> == '00' then return;

    case CurrentInstrSet() of
        when InstrSet_ARM
            step = 4;
        when InstrSet_Thumb, InstrSet_ThumbEE
            step = 2;
        when InstrSet_Jazelle
            step = 1;
    length = length / step;

    vcr_match = FALSE;
    brp_match = FALSE;

    // Each unit of the instruction is checked against the VCR and the BRPs. VCRMatch()
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    // Each unit of the instruction is checked against the VCR and the BRPs. VCRMatch()
    // and BRPMatch() might return UNKNOWN for units other than the first unit of the
    // instruction, as in some cases the generation of Debug events is UNPREDICTABLE.
    for W = 0 to length-1
        vcr_match = VCRMatch(address, W == 0) || vcr_match;

        // This code does not take into account the case where a mismatch breakpoint
        // does not match the address of an instruction but another breakpoint or
        // vector catch does match the instruction. In that situation, generation of
        // the Debug event is UNPREDICTABLE.
        for N = 0 to UInt(DBGDIDR.BRPs)
            brp_match = BRPMatch(N, address, W == 0) || brp_match;

        address = address + step;

    // A suitable debug event occurs if there has been a BRP match or a VCR match. If
    // both have occurred, just one debug event occurs, and its type is IMPLEMENTATION
    // DEFINED.
    if vcr_match || brp_match then
        if !vcr_match then BreakpointDebugEvent();
        elsif !brp_match then VectorCatchDebugEvent();
        else IMPLEMENTATION_DEFINED either BreakpointDebugEvent() or VectorCatchDebugEvent();

    return;

// BRPMatch()
// ==========

boolean BRPMatch(integer N, bits(32) address, boolean first)
    assert N <= UInt(DBGDIDR.BRPs);

    // If this breakpoint is not enabled, return immediately.
    if DBGBCR[N]<0> == '0' return FALSE;

    unk_match = FALSE;

    // Mode control match
    case DBGBCR[N]<2:1> of
        when '00'
            if UInt(DBGDIDR.Version) < 3 then
                UNPREDICTABLE;
            else
                case CPSR.M of
                    when '10000'  mode_control_match = TRUE;   // User mode
                    when '10011'  mode_control_match = TRUE;   // Supervisor mode
                    when '11111'  mode_control_match = TRUE;   // System mode
                    otherwise     mode_control_match = FALSE;  // Any other mode
        when '01'  mode_control_match = CurrentModeIsPrivileged();   // Privileged mode
        when '10'  mode_control_match = !CurrentModeIsPrivileged();  // Unprivileged mode
        when '11'  mode_control_match = TRUE;                        // Any mode

    // Byte lane select
    case CurrentInstrSet() of
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
    // and BRPMatch() might return UNKNOWN for units other than the first unit of the
    // instruction, as in some cases the generation of Debug events is UNPREDICTABLE.
    for W = 0 to length-1
        vcr_match = VCRMatch(address, W == 0) || vcr_match;

        // This code does not take into account the case where a mismatch breakpoint
        // does not match the address of an instruction but another breakpoint or
        // vector catch does match the instruction. In that situation, generation of
        // the Debug event is UNPREDICTABLE.
        for N = 0 to UInt(DBGDIDR.BRPs)
            brp_match = BRPMatch(N, address, W == 0) || brp_match;

        address = address + step;

    // A suitable debug event occurs if there has been a BRP match or a VCR match. If
    // both have occurred, just one debug event occurs, and its type is IMPLEMENTATION
    // DEFINED.
    if vcr_match || brp_match then
        if !vcr_match then BreakpointDebugEvent();
        elsif !brp_match then VectorCatchDebugEvent();
        else IMPLEMENTATION_DEFINED either BreakpointDebugEvent() or VectorCatchDebugEvent();

    return;

// BRPMatch()
// ==========

boolean BRPMatch(integer N, bits(32) address, boolean first)
    assert N <= UInt(DBGDIDR.BRPs);

    // If this breakpoint is not enabled, return immediately.
    if DBGBCR[N]<0> == '0' return FALSE;

    unk_match = FALSE;

    // Mode control match
    case DBGBCR[N]<2:1> of
        when '00'
            if UInt(DBGDIDR.Version) < 3 then
                UNPREDICTABLE;
            else
                case CPSR.M of
                    when '10000'  mode_control_match = TRUE;   // User mode
                    when '10011'  mode_control_match = TRUE;   // Supervisor mode
                    when '11111'  mode_control_match = TRUE;   // System mode
                    otherwise     mode_control_match = FALSE;  // Any other mode
        when '01'  mode_control_match = CurrentModeIsPrivileged();   // Privileged mode
        when '10'  mode_control_match = !CurrentModeIsPrivileged();  // Unprivileged mode
        when '11'  mode_control_match = TRUE;                        // Any mode

    // Byte lane select
    case CurrentInstrSet() of
        when InstrSet_ARM
            byte_select_match = (DBGBCR[N]<8:5> != '0000');
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    when InstrSet_ARM
        byte_select_match = (DBGBCR[N]<8:5> != '0000');
    when InstrSet_Thumb, InstrSet_ThumbEE
        case address<1> of
            when '0'   byte_select_match = (DBGBCR[N]<6:5> != '00');
            when '1'   byte_select_match = (DBGBCR[N]<8:7> != '00');
    when InstrSet_Jazelle
        case address<1:0> of
            when '00'  byte_select_match = (DBGBCR[N]<5> == '1');
            when '01'  byte_select_match = (DBGBCR[N]<6> == '1');
            when '10'  byte_select_match = (DBGBCR[N]<7> == '1');
            when '11'  byte_select_match = (DBGBCR[N]<8> == '1');

// Address mask
case DBGBCR[N]<28:24> of
    when '00000'
        // This implies no mask, but the byte address is always dealt with by
        // byte_select_match, so the mask always has the bottom two bits set.
        mask = ZeroExtend('11', 32);
    when '00001', '00010'
        UNPREDICTABLE;
    otherwise
        mask = ZeroExtend(Ones(UInt(DBGBCR[N]<28:24>)), 32);
        if DBGBCR[N]<8:5> != '1111' then unk_match = TRUE;

// Meaning of BVR
case DBGBCR[N]<22:20> of
    when '000'  // Unlinked IVA match
        cmp_in = address;  linked = FALSE;  mismatch = FALSE;  mon_debug_ok = TRUE;

    when '001'  // Linked IVA match
        cmp_in = address;  linked = TRUE;  mismatch = FALSE;  mon_debug_ok = TRUE;

    when '010'  // Unlinked context ID match
        if N < UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs) then UNPREDICTABLE;
        if DBGBCR[N]<8:5> != '1111' || DBGBCR[N]<28:24> != '00000' then unk_match = TRUE;
        mask = Zeros(32);
        cmp_in = CONTEXTIDR;  linked = FALSE;  mismatch = FALSE;  mon_debug_ok = FALSE;

    when '011'  // Linked context ID match (does not match directly, only via link)
        return FALSE;

    when '100'  // Unlinked IVA mismatch
        if UInt(DBGDIDR.Version) < 2 then UNPREDICTABLE;
        cmp_in = address;  linked = FALSE;  mismatch = TRUE;  mon_debug_ok = FALSE;

    when '101'  // Linked IVA mismatch
        if UInt(DBGDIDR.Version) < 2 then UNPREDICTABLE;
        cmp_in = address;  linked = TRUE;  mismatch = TRUE;  mon_debug_ok = FALSE;

    otherwise   // Reserved
        unk_match = TRUE;

if !IsZero(DBGBVR[N] AND mask) then unk_match = TRUE;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

```
    when InstrSet_Thumb, InstrSet_ThumbEE
        case address<1> of
            when '0'   byte_select_match = (DBGBCR[N]<6:5> != '00');
            when '1'   byte_select_match = (DBGBCR[N]<8:7> != '00');
    when InstrSet_Jazelle
        case address<1:0> of
            when '00'  byte_select_match = (DBGBCR[N]<5> == '1');
            when '01'  byte_select_match = (DBGBCR[N]<6> == '1');
            when '10'  byte_select_match = (DBGBCR[N]<7> == '1');
            when '11'  byte_select_match = (DBGBCR[N]<8> == '1');

// Address mask
case DBGBCR[N]<28:24> of
    when '00000'
        // This implies no mask, but the byte address is always dealt with by
        // byte_select_match, so the mask always has the bottom two bits set.
        mask = ZeroExtend('11', 32);
    when '00001', '00010'
        UNPREDICTABLE;
    otherwise
        mask = ZeroExtend(Ones(UInt(DBGBCR[N]<28:24>)), 32);
        if DBGBCR[N]<8:5> != '1111' then unk_match = TRUE;

// Meaning of BVR
case DBGBCR[N]<22:20> of
    when '000'  // Unlinked IVA match
        cmp_in = address;  linked = FALSE;  mismatch = FALSE;  mon_debug_ok = TRUE;

    when '001'  // Linked IVA match
        cmp_in = address;  linked = TRUE;  mismatch = FALSE;  mon_debug_ok = TRUE;

    when '010'  // Unlinked context ID match
        if N < UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs) then UNPREDICTABLE;
        if DBGBCR[N]<8:5> != '1111' || DBGBCR[N]<28:24> != '00000' then unk_match = TRUE;
        mask = Zeros(32);
        cmp_in = CONTEXTIDR;  linked = FALSE;  mismatch = FALSE;  mon_debug_ok = FALSE;

    when '011'  // Linked context ID match (does not match directly, only via link)
        return FALSE;

    when '100'  // Unlinked IVA mismatch
        if UInt(DBGDIDR.Version) < 2 then UNPREDICTABLE;
        cmp_in = address;  linked = FALSE;  mismatch = TRUE;  mon_debug_ok = FALSE;

    when '101'  // Linked IVA mismatch
        if UInt(DBGDIDR.Version) < 2 then UNPREDICTABLE;
        cmp_in = address;  linked = TRUE;  mismatch = TRUE;  mon_debug_ok = FALSE;

    otherwise   // Reserved
        unk_match = TRUE;

if !IsZero(DBGBVR[N] AND mask) then unk_match = TRUE;
BVR_match = byte_select_match && (cmp_in AND NOT(mask)) == DBGBVR[N];
if mismatch then BVR_match = !BVR_match;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    BVR_match = byte_select_match && (cmp_in AND NOT(mask)) == DBGBVR[N];
    if mismatch then BVR_match = !BVR_match;

    // If this is not the first unit of the instruction and there is an address match, then
    // the breakpoint match is UNPREDICTABLE, except in the "single-step" case where it is a
    // mismatch breakpoint without a range set. If there is a match on the first unit of the
    // instruction, that will override the UNKNOWN case here. In the single-step case, matches
    // on the subsequent units of the instruction are ignored.
    if BVR_match && !first then
        if mismatch && DBGBCR[N]<28:24> == '00000' then  // Single-step case
            BVR_match = FALSE;
        else
            BVR_match = boolean UNKNOWN;

    // Security state
    case DBGBCR[N]<15:14> of
        when '00'  secure_state_match = TRUE;         // Any state (or no Security Extensions)
        when '01'  secure_state_match = !IsSecure();  // Non-secure only
        when '10'  secure_state_match = IsSecure();   // Secure only
        when '11'  UNPREDICTABLE;                     // Reserved

    match = mode_control_match && BVR_match && secure_state_match;

    // If linked, check the linked BRP.
    if linked then match = match && BRPLinkMatch(UInt(DBGBCR[N]<19:16>));
    elsif DBGBCR[N]<19:16> != '0000' then unk_match = TRUE;

    // When Monitor debug-mode is configured:
    // * some types of event are ignored in v6 Debug and v6.1 Debug in privileged modes
    // * some types of event are UNPREDICTABLE in v7 Debug.
    if !mon_debug_ok && DBGDSCR<15:14> == '10' then
        if UInt(DBGDIDR.Version) < 3 then
            if CurrentModeIsPrivileged() then return FALSE;
        else
            if DBGBCR[N]<2:1> == '01' || DBGBCR[N]<2:1> == '11' then UNPREDICTABLE;

    if unk_match then
        return boolean UNKNOWN;
    else
        return match;

// BRPLinkMatch()
// ==============

boolean BRPLinkMatch(integer M)
    assert M <= UInt(DBGDIDR.BRPs);

    if M < UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs) then UNPREDICTABLE;

    // If this breakpoint is not enabled, return immediately.
    if DBGBCR[M]<0> == '0' return FALSE;

    unk_match = FALSE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
    // If this is not the first unit of the instruction and there is an address match, then
    // the breakpoint match is UNPREDICTABLE, except in the "single-step" case where it is a
    // mismatch breakpoint without a range set. If there is a match on the first unit of the
    // instruction, that will override the UNKNOWN case here. In the single-step case, matches
    // on the subsequent units of the instruction are ignored.
    if BVR_match && !first then
        if mismatch && DBGBCR[N]<28:24> == '00000' then  // Single-step case
            BVR_match = FALSE;
        else
            BVR_match = boolean UNKNOWN;

    // Security state
    case DBGBCR[N]<15:14> of
        when '00'  secure_state_match = TRUE;         // Any state (or no Security Extensions)
        when '01'  secure_state_match = !IsSecure();  // Non-secure only
        when '10'  secure_state_match = IsSecure();   // Secure only
        when '11'  UNPREDICTABLE;                     // Reserved

    match = mode_control_match && BVR_match && secure_state_match;

    // If linked, check the linked BRP.
    if linked then match = match && BRPLinkMatch(UInt(DBGBCR[N]<19:16>));
    elsif DBGBCR[N]<19:16> != '0000' then unk_match = TRUE;

    // When Monitor debug-mode is configured:
    // * some types of event are ignored in v6 Debug and v6.1 Debug in privileged modes
    // * some types of event are UNPREDICTABLE in v7 Debug.
    if !mon_debug_ok && DBGDSCR<15:14> == '10' then
        if UInt(DBGDIDR.Version) < 3 then
            if CurrentModeIsPrivileged() then return FALSE;
        else
            if DBGBCR[N]<2:1> == '01' || DBGBCR[N]<2:1> == '11' then UNPREDICTABLE;

    if unk_match then
        return boolean UNKNOWN;
    else
        return match;

// BRPLinkMatch()
// ==============

boolean BRPLinkMatch(integer M)
    assert M <= UInt(DBGDIDR.BRPs);

    if M < UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs) then UNPREDICTABLE;

    // If this breakpoint is not enabled, return immediately.
    if DBGBCR[M]<0> == '0' return FALSE;

    unk_match = FALSE;

    if DBGBCR[M]<2:1> != '11' then unk_match = TRUE;
    if DBGBCR[M]<8:5> != '1111' then unk_match = TRUE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    if DBGBCR[M]<2:1> != '11' then unk_match = TRUE;
    if DBGBCR[M]<8:5> != '1111' then unk_match = TRUE;
    if DBGBCR[M]<15:14> != '00' then unk_match = TRUE;
    if DBGBCR[M]<19:16> != '0000' then unk_match = TRUE;
    if DBGBCR[M]<22:20> != '011' then unk_match = TRUE;
    if DBGBCR[M]<28:24> != '00000' then unk_match = TRUE;

    if unk_match then
        return boolean UNKNOWN;
    else
        return (CONTEXTIDR == DBGBVR[M]);

// Variables used to record most recent interrupts of various types.

bits(32) VCR_Recent_IRQ_S;
bits(32) VCR_Recent_IRQ_NS;
bits(32) VCR_Recent_FIQ_S;
bits(32) VCR_Recent_FIQ_NS;
boolean VCR_Recent_IRQ_S_Valid;
boolean VCR_Recent_IRQ_NS_Valid;
boolean VCR_Recent_FIQ_S_Valid;
boolean VCR_Recent_FIQ_NS_Valid;


// VCR_OnTakingInterrupt()
// =======================

VCR_OnTakingInterrupt(bits(32) vector, boolean FIQnIRQ)
    if SCTLR.VE == '1' then
        if FIQnIRQ then
            if IsSecure() then
                if DBGVCR<7> == '0' || (HaveSecurityExt() && SCR.FIQ == '1') then
                    IMPLEMENTATION_DEFINED whether the variables are updated;
                else
                    VCR_Recent_FIQ_S = vector;
                    VCR_Recent_FIQ_S_Valid = TRUE;
            else
                if DBGVCR<31> == '0' || (HaveSecurityExt() && SCR.FIQ == '1') then
                    IMPLEMENTATION_DEFINED whether the variables are updated;
                else
                    VCR_Recent_FIQ_NS = vector;
                    VCR_Recent_FIQ_NS_Valid = TRUE;
        else
            if IsSecure() then
                if DBGVCR<6> == '0' || (HaveSecurityExt() && SCR.IRQ == '1') then
                    IMPLEMENTATION_DEFINED whether the variables are updated;
                else
                    VCR_Recent_IRQ_S = vector;
                    VCR_Recent_IRQ_S_Valid = TRUE;
            else
                if DBGVCR<30> == '0' ||  (HaveSecurityExt() && SCR.IRQ == '1')  then
                    IMPLEMENTATION_DEFINED whether the variables are updated;
                else
                    VCR_Recent_IRQ_NS = vector;
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

```
    if DBGBCR[M]<15:14> != '00' then unk_match = TRUE;
    if DBGBCR[M]<19:16> != '0000' then unk_match = TRUE;
    if DBGBCR[M]<22:20> != '011' then unk_match = TRUE;
    if DBGBCR[M]<28:24> != '00000' then unk_match = TRUE;

    if unk_match then
        return boolean UNKNOWN;
    else
        return (CONTEXTIDR == DBGBVR[M]);

// Variables used to record most recent interrupts of various types.

bits(32) VCR_Recent_IRQ_S;
bits(32) VCR_Recent_IRQ_NS;
bits(32) VCR_Recent_FIQ_S;
bits(32) VCR_Recent_FIQ_NS;
boolean VCR_Recent_IRQ_S_Valid;
boolean VCR_Recent_IRQ_NS_Valid;
boolean VCR_Recent_FIQ_S_Valid;
boolean VCR_Recent_FIQ_NS_Valid;


// VCR_OnTakingInterrupt()
// =======================

VCR_OnTakingInterrupt(bits(32) vector, boolean FIQnIRQ)
    if SCTLR.VE == '1' then
        if FIQnIRQ then
            if IsSecure() then
                if DBGVCR<7> == '0' || (HaveSecurityExt() && SCR.FIQ == '1') then
                    IMPLEMENTATION_DEFINED whether the variables are updated;
                else
                    VCR_Recent_FIQ_S = vector;
                    VCR_Recent_FIQ_S_Valid = TRUE;
            else
                if DBGVCR<31> == '0' || (HaveSecurityExt() && SCR.FIQ == '1') then
                    IMPLEMENTATION_DEFINED whether the variables are updated;
                else
                    VCR_Recent_FIQ_NS = vector;
                    VCR_Recent_FIQ_NS_Valid = TRUE;
        else
            if IsSecure() then
                if DBGVCR<6> == '0' || (HaveSecurityExt() && SCR.IRQ == '1') then
                    IMPLEMENTATION_DEFINED whether the variables are updated;
                else
                    VCR_Recent_IRQ_S = vector;
                    VCR_Recent_IRQ_S_Valid = TRUE;
            else
                if DBGVCR<30> == '0' ||  (HaveSecurityExt() && SCR.IRQ == '1')  then
                    IMPLEMENTATION_DEFINED whether the variables are updated;
                else
                    VCR_Recent_IRQ_NS = vector;
                    VCR_Recent_IRQ_NS_Valid = TRUE;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
                    VCR_Recent_IRQ_NS_Valid = TRUE;

    return;


// VCRVectorMatch()
// ================
//
// The result of this function says whether iaddr and eaddr match for vector catch purposes:
//   TRUE             if they definitely match
//   boolean UNKNOWN  if it is UNPREDICTABLE whether they match
//   FALSE            if they definitely do not match

boolean VCRVectorMatch(bits(32) iaddr, boolean first, bits(32) eaddr)

    match = FALSE;
    unpred = FALSE;

    if eaddr<31:2> == iaddr<31:2> then
        if eaddr<1:0> == iaddr<1:0> then
            // Exact address match is a definite match if on the first unit of the instruction,
            // otherwise an UNPREDICTABLE match.
            if first then match = TRUE;  else unpred = TRUE;
        else
            // Check for other cases of UNPREDICTABLE matches.
            case CurrentInstrSet() of
                when InstrSet_ARM
                    unpred = TRUE;
                when InstrSet_Thumb, InstrSet_ThumbEE
                    if iaddr<1> == eaddr<1> then unpred = TRUE;
                    if iaddr<1:0> == '10' && eaddr<1:0> == '00' then unpred = TRUE;
                when InstrSet_Jazelle
                    if eaddr<1:0> == '00' then unpred = TRUE;
                    if eaddr<1:0> == '10' && iaddr<1:0> == '11' then unpred = TRUE;

    if match then
        return TRUE;
    elsif unpred then
        return boolean UNKNOWN;
    else
        return FALSE;

// VCRMatch()
// ==========

boolean VCRMatch(bits(32) address, boolean first)

    // Determine addresses for IRQ and FIQ comparisons.

   if SCTLR.VE == '0' then
       VCR_Recent_IRQ_S_Valid = FALSE;  VCR_Recent_IRQ_NS_Valid = FALSE;
       VCR_Recent_FIQ_S_Valid = FALSE;  VCR_Recent_FIQ_NS_Valid = FALSE;
       irq_addr = ExcVectorBase() + 24;  irq_addr_v = TRUE;
       fiq_addr = ExcVectorBase() + 28;  fiq_addr_v = TRUE;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
    return;


// VCRVectorMatch()
// ================
//
// The result of this function says whether iaddr and eaddr match for vector catch purposes:
//   TRUE             if they definitely match
//   boolean UNKNOWN  if it is UNPREDICTABLE whether they match
//   FALSE            if they definitely do not match

boolean VCRVectorMatch(bits(32) iaddr, boolean first, bits(32) eaddr)

    match = FALSE;
    unpred = FALSE;

    if eaddr<31:2> == iaddr<31:2> then
        if eaddr<1:0> == iaddr<1:0> then
            // Exact address match is a definite match if on the first unit of the instruction,
            // otherwise an UNPREDICTABLE match.
            if first then match = TRUE;  else unpred = TRUE;
        else
            // Check for other cases of UNPREDICTABLE matches.
            case CurrentInstrSet() of
                when InstrSet_ARM
                    unpred = TRUE;
                when InstrSet_Thumb, InstrSet_ThumbEE
                    if iaddr<1> == eaddr<1> then unpred = TRUE;
                    if iaddr<1:0> == '10' && eaddr<1:0> == '00' then unpred = TRUE;
                when InstrSet_Jazelle
                    if eaddr<1:0> == '00' then unpred = TRUE;
                    if eaddr<1:0> == '10' && iaddr<1:0> == '11' then unpred = TRUE;

    if match then
        return TRUE;
    elsif unpred then
        return boolean UNKNOWN;
    else
        return FALSE;

// VCRMatch()
// ==========

boolean VCRMatch(bits(32) address, boolean first)

    // Determine addresses for IRQ and FIQ comparisons.

   if SCTLR.VE == '0' then
       VCR_Recent_IRQ_S_Valid = FALSE;  VCR_Recent_IRQ_NS_Valid = FALSE;
       VCR_Recent_FIQ_S_Valid = FALSE;  VCR_Recent_FIQ_NS_Valid = FALSE;
       irq_addr = ExcVectorBase() + 24;  irq_addr_v = TRUE;
       fiq_addr = ExcVectorBase() + 28;  fiq_addr_v = TRUE;
    else
        if IsSecure() then
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
else
    if IsSecure() then
        irq_addr = VCR_Recent_IRQ_S;  irq_addr_v = VCR_Recent_IRQ_S_Valid;
        fiq_addr = VCR_Recent_FIQ_S;  fiq_addr_v = VCR_Recent_FIQ_S_Valid;
    else
        irq_addr = VCR_Recent_IRQ_NS;  irq_addr_v = VCR_Recent_IRQ_NS_Valid;
        fiq_addr = VCR_Recent_FIQ_NS;  fiq_addr_v = VCR_Recent_FIQ_NS_Valid;


a_match = FALSE;  // Boolean for a match on an abort vector
match = FALSE;    // Boolean for a match on any other vector

// Check for non-monitor, non-reset matches, using DBGVCR<7:1> if no Security
// Extensions or in Secure state, or DBGVCR<31:25> if in Non-secure state.
start = if IsSecure() then 0 else 24;
if DBGVCR<start+1> == '1' then
    match = match || VCRVectorMatch(address, first, ExcVectorBase()+4);
if DBGVCR<start+2> == '1' then
    match = match || VCRVectorMatch(address, first, ExcVectorBase()+8);
if DBGVCR<start+3> == '1' then
    a_match = a_match || VCRVectorMatch(address, first, ExcVectorBase()+12);
if DBGVCR<start+4> == '1' then
    a_match = a_match || VCRVectorMatch(address, first, ExcVectorBase()+16);
if DBGVCR<start+6> == '1' then
    if HaveSecurityExt() && SCR.IRQ == '1' && SCTLR.VE == '1' then
        IMPLEMENTATION_DEFINED what test is made, if any;
    else if irq_addr_v then
        match = match || VCRVectorMatch(address, first, irq_addr);
if DBGVCR<start+7> == '1' then
    if HaveSecurityExt() && SCR.FIQ == '1' && SCTLR.VE == '1' then
        IMPLEMENTATION_DEFINED what test is made, if any;
    else if fiq_addr_v then
        match = match || VCRVectorMatch(address, first, fiq_addr);

// If we have the Security Extensions and are in Secure state, check for monitor matches.
if HaveSecurityExt() && IsSecure() then
    if DBGVCR<10> == '1' then
        match = match || VCRVectorMatch(address, first, MVBAR+8);
    if DBGVCR<11> == '1' then
        a_match = a_match || VCRVectorMatch(address, first, MVBAR+12);
    if DBGVCR<12> == '1' then
        a_match = a_match || VCRVectorMatch(address, first, MVBAR+16);
    if DBGVCR<14> == '1' then
        match = match || VCRVectorMatch(address, first, MVBAR+24);
    if DBGVCR<15> == '1' then
        match = match || VCRVectorMatch(address, first, MVBAR+28);

// Check for reset matches.
// In v7 Debug this check is made regardless of the security state.
// In v6 Debug and v6.1 Debug this check is only made in Secure state.
vector = if SCTLR.V == '1' then Ones(16):Zeros(16) else Zeros(32);
if DBGVCR<0> == '1' && (UInt(DBGDIDR.Version) >= 3 || IsSecure()) then
    match = match || VCRVectorMatch(address, first, vector);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

```
        irq_addr = VCR_Recent_IRQ_S;  irq_addr_v = VCR_Recent_IRQ_S_Valid;
        fiq_addr = VCR_Recent_FIQ_S;  fiq_addr_v = VCR_Recent_FIQ_S_Valid;
    else
        irq_addr = VCR_Recent_IRQ_NS;  irq_addr_v = VCR_Recent_IRQ_NS_Valid;
        fiq_addr = VCR_Recent_FIQ_NS;  fiq_addr_v = VCR_Recent_FIQ_NS_Valid;


a_match = FALSE;  // Boolean for a match on an abort vector
match = FALSE;    // Boolean for a match on any other vector

// Check for non-monitor, non-reset matches, using DBGVCR<7:1> if no Security
// Extensions or in Secure state, or DBGVCR<31:25> if in Non-secure state.
start = if IsSecure() then 0 else 24;
if DBGVCR<start+1> == '1' then
    match = match || VCRVectorMatch(address, first, ExcVectorBase()+4);
if DBGVCR<start+2> == '1' then
    match = match || VCRVectorMatch(address, first, ExcVectorBase()+8);
if DBGVCR<start+3> == '1' then
    a_match = a_match || VCRVectorMatch(address, first, ExcVectorBase()+12);
if DBGVCR<start+4> == '1' then
    a_match = a_match || VCRVectorMatch(address, first, ExcVectorBase()+16);
if DBGVCR<start+6> == '1' then
    if HaveSecurityExt() && SCR.IRQ == '1' && SCTLR.VE == '1' then
        IMPLEMENTATION_DEFINED what test is made, if any;
    else if irq_addr_v then
        match = match || VCRVectorMatch(address, first, irq_addr);
if DBGVCR<start+7> == '1' then
    if HaveSecurityExt() && SCR.FIQ == '1' && SCTLR.VE == '1' then
        IMPLEMENTATION_DEFINED what test is made, if any;
    else if fiq_addr_v then
        match = match || VCRVectorMatch(address, first, fiq_addr);

// If we have the Security Extensions and are in Secure state, check for monitor matches.
if HaveSecurityExt() && IsSecure() then
    if DBGVCR<10> == '1' then
        match = match || VCRVectorMatch(address, first, MVBAR+8);
    if DBGVCR<11> == '1' then
        a_match = a_match || VCRVectorMatch(address, first, MVBAR+12);
    if DBGVCR<12> == '1' then
        a_match = a_match || VCRVectorMatch(address, first, MVBAR+16);
    if DBGVCR<14> == '1' then
        match = match || VCRVectorMatch(address, first, MVBAR+24);
    if DBGVCR<15> == '1' then
        match = match || VCRVectorMatch(address, first, MVBAR+28);

// Check for reset matches.
// In v7 Debug this check is made regardless of the security state.
// In v6 Debug and v6.1 Debug this check is only made in Secure state.
vector = if SCTLR.V == '1' then Ones(16):Zeros(16) else Zeros(32);
if DBGVCR<0> == '1' && (UInt(DBGDIDR.Version) >= 3 || IsSecure()) then
    match = match || VCRVectorMatch(address, first, vector);

// When Monitor debug-mode is configured, abort vector catches are ignored in v6 Debug
// and v6.1 Debug, but UNPREDICTABLE in v7 Debug.
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    // When Monitor debug-mode is configured, abort vector catches are ignored in v6 Debug
    // and v6.1 Debug, but UNPREDICTABLE in v7 Debug.
    if a_match && DBGDSCR<15:14> == '10' then
        if UInt(DBGDIDR.Version) < 3 then
            a_match = FALSE;
        else
            UNPREDICTABLE;

    return match || a_match;
```

Watchpoints

If invasive debug is enabled, the `Debug_CheckDataAccess()` function checks WRP matches for each data access. If the implementation includes IMPLEMENTATION DEFINED support for watchpoint generation on memory hint operations, or on cache maintenance operations, the function also checks for WRP matches on the appropriate operations. If a match is found the function calls `WatchpointDebugEvent()`. If the debug event is not permitted, it is ignored by the processor.

On a simple sequential execution model:

- for a synchronous watchpoint, the `Debug_CheckDataAccess()` test is made before the data access
- for an asynchronous watchpoint, the `Debug_CheckDataAccess()` test is made after the data access.

For more information see *Synchronous and Asynchronous Watchpoint debug events* on page C3-18.

The `WRPMatch()` function checks an individual WRP match. In ARMv7, it is IMPLEMENTATION DEFINED whether WRP matches use eight byte lanes or four. The `WRPUsesEightByteLanes()` function returns TRUE if they use eight byte lanes and FALSE if they use four. Using eight byte lanes is permitted only in ARMv7.

```
boolean WRPUsesEightByteLanes()
```

For these functions the parameters `read`, `write`, `privileged` and `secure` are determined at the point the access is made, and not from the state of the processor at the point where `WRPMatch()` is executed. For swaps, `read` = `write` = TRUE.

```
// Debug_CheckDataAccess()
// =======================

boolean Debug_CheckDataAccess(bits(32) address, integer size, boolean read,
                              boolean write, boolean privileged, boolean secure)

    // Do nothing if debug disabled;
    if DBGDSCR<15:14> == '00' then return;

    match = FALSE;
    // Each byte accessed by the data access is checked
    for byte = address to address + size - 1
        for N = 0 to UInt(DBGDIDR.WRPs)
            if WRPMatch(N, byte, read, write, privileged, secure) then match = TRUE;

    if match then WatchpointDebugEvent();
    return;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
    if a_match && DBGDSCR<15:14> == '10' then
        if UInt(DBGDIDR.Version) < 3 then
            a_match = FALSE;
        else
            UNPREDICTABLE;

    return match || a_match;
```

ウォッチポイント

侵襲性デバッグが可能な場合、各データアクセスについて、`Debug_CheckDataAccess()` 関数により WRP 一致がチェックされます。実装に、メモリヒント操作またはキャッシュ保守操作におけるウォッチポイント生成の実装定義のサポートが含まれている場合、この関数は対応する操作についての WRP 一致もチェックします。一致が見つかった場合、この関数により `WatchpointDebugEvent()` が呼び出されます。デバッグイベントが許可されていない場合、プロセッサにより無視されます。

単純なシーケンシャルの実行モデルでは、次の動作が行われます。

- 同期ウォッチポイントの場合、`Debug_CheckDataAccess()` テストはデータアクセスの前に行われます。
- 非同期ウォッチポイントの場合、`Debug_CheckDataAccess()` テストはデータアクセスの後に行われます。

詳細については、*同期と非同期のウォッチポイントデバッグ イベント* P. C3- 18 を参照して下さい。

`WRPMatch()` 関数は、個別の WRP 一致をチェックします。ARMv7 では、WRP 一致に 8 バイトレーンと 4 バイトレーンのどちらを使用するかは実装定義です。`WRPUsesEightByteLanes()` 関数は、8 バイトレーンが使用されている場合は TRUE、4 バイトレーンが使用されている場合は FALSE を返します。8 バイトレーンの使用は、ARMv7 でのみ許可されています。

```
boolean WRPUsesEightByteLanes()
```

これらの関数について、`read`、`write`、`privileged`、`secure` の各パラメータは、`WRPMatch()` が実行された時点でのプロセッサの状態からではなく、アクセスが行われた時点で決定されます。スワップの場合、`read` = `write` = TRUE です。

```
// Debug_CheckDataAccess()

// =======================


boolean Debug_CheckDataAccess(bits(32) address, integer size, boolean read,

                              boolean write, boolean privileged, boolean secure)


    // Do nothing if debug disabled;

    if DBGDSCR<15:14> == '00' then return;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
// WRPMatch()
// ==========

boolean WRPMatch(integer N, bits(32) address, boolean read, boolean write,
                 boolean privileged, boolean secure)
    assert N <= UInt(DBGDIDR.WRPs);

    // If watchpoint is not enabled, return immediately.
    if DBGWCR[N]<0> == '0' return FALSE;

    // Access privilege match
    case DBGWCR[N]<2:1> of
        when '00'  UNPREDICTABLE;                  // Reserved
        when '01'  privilege_match = privileged;   // Only privileged accesses
        when '10'  privilege_match = !privileged;  // Only unprivileged accesses
        when '11'  privilege_match = TRUE;         // Any access

    // Load/Store access control match
    case DBGWCR[N]<4:3> of
        when '00'  UNPREDICTABLE;                  // Reserved
        when '01'  load_store_match = read;        // Only load, load exclusive or swap
        when '10'  load_store_match = write;       // Only store, store exclusive or swap
        when '11'  load_store_match = TRUE;        // All accesses

    // Address match
    case DBGWCR[N]<28:24> of
        when '00000'  // No mask
            // If implementation uses 8 byte lanes, DBGWVR[N]<2> == '1' selects 4 byte lane
            // behavior.
            if DBGWVR[N]<2> == '1' then
                bits = 2;
                if DBGWCR[N]<12:9> != '0000' then UNPREDICTABLE;
            else
                bits = if WRPUsesEightByteLanes() then 3 else 2;
            mask = ZeroExtend(Ones(bits), 32);
            if !IsZero(DBGWVR[N]<1:0>) then UNPREDICTABLE;
            byte = UInt(address<bits-1:0>);
            WVR_match = ((address AND NOT(mask)) == DBGWVR[N]) && (DBGWCR[N]<5+byte> == '1');

        when '00001', '00010'  // Reserved
            UNPREDICTABLE;

        otherwise  // Masked address check
            mask = ZeroExtend(Ones(UInt(DBGWCR[N]<28:24>)), 32);
            if !IsZero(DBGWVR[N] AND mask) then UNPREDICTABLE;
            if DBGWCR[N]<8:5> != '1111' then UNPREDICTABLE;
            if WRPUsesEightByteLanes() && (DBGWCR[N]<12:9> != '1111') then UNPREDICTABLE;
            WVR_match = ((address AND NOT(mask)) == DBGWVR[N]);

    // Security state
    case DBGWCR[N]<15:14> of
        when '00'  secure_state_match = TRUE;     // Any access (or no Security Extensions)
        when '01'  secure_state_match = !secure;  // Only non-secure accesses
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

```
    match = FALSE;

    // Each byte accessed by the data access is checked

    for byte = address to address + size - 1

        for N = 0 to UInt(DBGDIDR.WRPs)

            if WRPMatch(N, byte, read, write, privileged, secure) then match = TRUE;


    if match then WatchpointDebugEvent();

    return;
```


```
// WRPMatch()
// ==========

boolean WRPMatch(integer N, bits(32) address, boolean read, boolean write,
                 boolean privileged, boolean secure)
    assert N <= UInt(DBGDIDR.WRPs);

    // If watchpoint is not enabled, return immediately.
    if DBGWCR[N]<0> == '0' return FALSE;

    // Access privilege match
    case DBGWCR[N]<2:1> of
        when '00'  UNPREDICTABLE;                  // Reserved
        when '01'  privilege_match = privileged;   // Only privileged accesses
        when '10'  privilege_match = !privileged;  // Only unprivileged accesses
        when '11'  privilege_match = TRUE;         // Any access

    // Load/Store access control match
    case DBGWCR[N]<4:3> of
        when '00'  UNPREDICTABLE;                  // Reserved
        when '01'  load_store_match = read;        // Only load, load exclusive or swap
        when '10'  load_store_match = write;       // Only store, store exclusive or swap
        when '11'  load_store_match = TRUE;        // All accesses

    // Address match
    case DBGWCR[N]<28:24> of
        when '00000'  // No mask
            // If implementation uses 8 byte lanes, DBGWVR[N]<2> == '1' selects 4 byte lane
            // behavior.
            if DBGWVR[N]<2> == '1' then
                bits = 2;
                if DBGWCR[N]<12:9> != '0000' then UNPREDICTABLE;
            else
                bits = if WRPUsesEightByteLanes() then 3 else 2;
            mask = ZeroExtend(Ones(bits), 32);
            if !IsZero(DBGWVR[N]<1:0>) then UNPREDICTABLE;
            byte = UInt(address<bits-1:0>);
            WVR_match = ((address AND NOT(mask)) == DBGWVR[N]) && (DBGWCR[N]<5+byte> == '1');
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
    when '10'  secure_state_match = secure;   // Only secure accesses
    when '11'  UNPREDICTABLE;                 // Reserved

match = privilege_match && load_store_match && WVR_match && secure_state_match;

// Check for linking
linked = (DBGWCR[N]<22> == '1');
if linked then match = match && BRPLinkMatch(UInt(DBGWCR[N]<19:16>));
elsif DBGWCR[N]<19:16> != '0000' then UNPREDICTABLE;

return match;
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
    when '00001', '00010'  // Reserved
        UNPREDICTABLE;

    otherwise  // Masked address check
        mask = ZeroExtend(Ones(UInt(DBGWCR[N]<28:24>)), 32);
        if !IsZero(DBGWVR[N] AND mask) then UNPREDICTABLE;
        if DBGWCR[N]<8:5> != '1111' then UNPREDICTABLE;
        if WRPUsesEightByteLanes() && (DBGWCR[N]<12:9> != '1111') then UNPREDICTABLE;
        WVR_match = ((address AND NOT(mask)) == DBGWVR[N]);

// Security state
case DBGWCR[N]<15:14> of
    when '00'  secure_state_match = TRUE;     // Any access (or no Security Extensions)
    when '01'  secure_state_match = !secure;  // Only non-secure accesses
    when '10'  secure_state_match = secure;   // Only secure accesses
    when '11'  UNPREDICTABLE;                 // Reserved

match = privilege_match && load_store_match && WVR_match && secure_state_match;

// Check for linking
linked = (DBGWCR[N]<22> == '1');
if linked then match = match && BRPLinkMatch(UInt(DBGWCR[N]<19:16>));
elsif DBGWCR[N]<19:16> != '0000' then UNPREDICTABLE;

return match;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C3.3 Halting debug events

A Halting debug event is one of the following:

- An External Debug Request debug event. This is a request from the system for the processor to enter Debug state.

  The method of generating an External Debug Request is IMPLEMENTATION DEFINED. Typically it is by asserting an External Debug Request input to the processor.

- A Halt Request debug event. This occurs when the debug logic receives a Halt request command.

  In v7 Debug, a debugger generates a Halt request command by writing 1 to the DBGDRCR Halt request bit, see *Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29.

- An OS Unlock Catch debug event. This occurs when both of the following are true:
  — the OS Unlock Catch is enabled in the Event Catch Register
  — the OS Lock transitions from the locked to the unlocked condition.

  For details see *Event Catch Register (DBGECR)* on page C10-78 and *OS Lock Access Register (DBGOSLAR)* on page C10-75.

If invasive debug is disabled when one of these events is detected, the request is ignored and no Halting debug event occurs. Invasive debug is disabled when the external debug interface signal **DBGEN** is LOW.

If **DBGEN** is HIGH, meaning that invasive debug is enabled, and a Halting debug event occurs when it is not permitted, the Halting debug event is pended. This means that the processor enters Debug state when it transitions to a security state or processor mode where the Halting debug event is permitted.

However, if **DBGEN** goes LOW before the processor enters the security state or processor mode where the Halting debug event is permitted, it is UNPREDICTABLE whether the event remains pended. If the debug logic is reset before the processor enters the permitted security state or processor mode, the processor must remove pending Halt Request and OS Unlock catch debug events. Whether a pending External Debug Request debug event is removed is IMPLEMENTATION DEFINED.

Note

The IMPLEMENTATION DEFINED details of External Debug Request might specify that it is pended externally by the peripheral that is driving it until the processor acknowledges the request by entering Debug state. In such a system the pending request is typically held over a debug logic reset.

If a Halting debug event occurs when debug is enabled and the event is permitted, or the Halting debug event becomes permitted while it is pending, it is guaranteed that Debug state is entered by the end of the next *Instruction Synchronization Barrier* (ISB) operation, exception entry, or exception return.

See *Run-control and cross-triggering signals* on page AppxA-5 for details of the recommended external debug interface.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C3.3 ホールトデバッグ イベント

ホールトデバッグ イベントは次のいずれかです。

- 外部デバッグ要求デバッグイベント。これは、プロセッサがデバッグ状態に遷移することをシステムから要求するものです。

  外部デバッグ要求を生成する方式は実装定義です。一般には、プロセッサへの外部デバッグ要求入力をアサートすることにより行われます。

- ホールト要求デバッグイベント。これは、デバッグロジックがホールト要求コマンドを受信したときに発生します。

  v7 デバッグでは、デバッガは DBGDRCR ホールト要求ビットに 1 を書き込むことでホールト要求コマンドを生成します。*デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10- 29 を参照して下さい。

- OS ロック解除キャッチデバッグ イベント。これは、次の条件の両方が TRUE の場合に発生します。
  — イベントキャッチ レジスタで OS ロック解除キャッチが可能である。
  — OS ロックが、ロック状態からロック解除状態に遷移した。

  詳細については、*イベントキャッチ レジスタ (DBGECR)* P. C10- 78 および *OS ロックアクセス レジスタ (DBGOSLAR)* P. C10- 75 を参照して下さい。

これらのイベントの 1 つが検出されたときに侵襲性デバッグが不可能な場合、要求は無視され、ホールトデバッグ イベントは発生しません。外部デバッグインタフェースの信号 DBGEN が LOW の場合、侵襲性デバッグは不可能です。

DBGEN が HIGH の場合、侵襲性デバッグが可能なことを示し、ホールトデバッグ イベントが許可されていないときにホールトデバッグ イベントが発生すると、そのイベントは保留されます。この場合、ホールトデバッグ イベントが許可されるセキュリティ状態またはプロセッサモードへの遷移が発生したとき、プロセッサはデバッグ状態に遷移します。

ただし、ホールトデバッグ イベントが許可されるセキュリティ状態またはプロセッサモードにプロセッサが遷移する前に DBGEN が LOW になった場合、イベントが保留のまま維持されるかどうかは予測不能です。プロセッサが許可されるセキュリティ状態やプロセッサモードに遷移する前に、デバッグロジックがリセットされた場合、プロセッサは保留中のホールト要求および OS ロック解除キャッチデバッグ イベントを削除する必要があります。保留中の外部デバッグ要求デバッグイベントが削除されるかどうかは実装定義です。

注

外部デバッグ要求の実装定義の詳細では、このような保留中の要求が、プロセッサがデバッグ状態に遷移して要求に応答するまで、要求を駆動するペリフェラルによって外部的に保留されることが指定されている場合があります。このようなシステムでは一般に、保留中の要求はデバッグロジックがリセットされても保持されます。

デバッグが可能でイベントが許可されているときにホールトデバッグ イベントが発生した場合、またはホールトデバッグ イベントが保留中のときに許可された場合、次の*命令同期バリア* (ISB) 操作、例外開始、または例外からの復帰の終了時に必ずデバッグ状態に遷移します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In v6 Debug and v6.1 Debug:

- if the processor implements the recommended ARM Debug Interface v4, the Halt request command is issued through the JTAG interface, by placing the HALT instruction in the IR and taking the *Debug Test Access Port State Machine* (Debug TAP State Machine) through the Run-Test/Idle state
- the OS Unlock Catch debug event is not supported.

In v6 Debug it is IMPLEMENTATION DEFINED whether Halting debug events cause entry to Debug state when Halting debug-mode is not configured and enabled.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

推奨される外部デバッグインタフェースの詳細については、*実行制御およびクロストリガ信号* P. 付録 A-5 を参照して下さい。

v6 デバッグおよび v6.1 デバッグでは、次の条件が適用されます。

- プロセッサに、推奨の ARM デバッグインタフェース v4 が実装されている場合、ホールト要求コマンドは JTAG インタフェースから、IR の HALT 命令を置き換え、実行 - テスト / アイドル状態を経由して*デバッグテストアクセスポート ステートマシン*（デバッグ TAP ステートマシン）を取得することにより発行されます。
- OS ロック解除キャッチデバッグ イベントはサポートされません。

v6 デバッグでは、ホールトデバッグ モードが構成され可能になっていない場合、ホールトデバッグ イベントによってデバッグ状態への遷移が引き起こされるかどうかは実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C3.4 Generation of debug events

The generation of Breakpoint and Watchpoint debug events can be dependent on the context of the processor, including:

- the current processor mode
- the contents of the CONTEXTIDR
- the Secure security state setting, if the processor implements Security Extensions.

The generation of debug events is also dependent on the state of the debug event generation logic:

- Breakpoint debug events are dependent on the contents of the relevant Breakpoint Register Pair (BRP)
- Watchpoint debug events are dependent on the contents of the relevant Watchpoint Register Pair (WRP)
- Linked Breakpoint or Watchpoint debug events are dependent on the settings of a second BRP
- Vector Catch debug events are dependent on the settings in the Vector Catch Register (DBGVCR)
- OS Unlock Catch debug events are dependent on the setting of the Event Catch Register (DBGECR).

In addition, as shown in Table C3-1 on page C3-2, the generation of debug events is dependent on:

- the invasive debug authentication settings, see Chapter C2 *Invasive Debug Authentication*
- the values of the DBGDSCR.HDBGen and DBGDSCR.MDBGen bits, see *Debug Status and Control Register (DBGDSCR)* on page C10-10.

The following events are guaranteed to take effect on the debug event generation logic by the end of the next ISB operation, exception entry, or exception return:

- Context changing operations, including:
  - — mode changes
  - — writes to the CONTEXTIDR
  - — security state changes.
- Operations that change the state of the debug event generation logic, including:
  - — writes to BRP registers, for Breakpoint debug events, or Linked Breakpoint or Watchpoint debug events
  - — writes to WRP registers, for Watchpoint debug events
  - — writes to the DBGVCR, for Vector Catch debug events
  - — writes to the DBGECR, for OS Unlock Catch debug events
  - — changes to the authentication signals
  - — writes to the DBGDSCR.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C3.4 デバッグ イベントの生成

ブレークポイントおよびウォッチポイントデバッグ イベントの生成は、次のものを含むプロセッサのコンテキストに依存する可能性があります。

- 現在のプロセッサモード
- CONTEXTIDR の内容
- プロセッサにセキュリティ拡張機能が実装されている場合、セキュアセキュリティ状態の設定

デバッグイベントの生成は、デバッグイベント生成ロジックの次のような状態にも依存します。

- ブレークポイントデバッグ イベントは、関連するブレークポイントレジスタ ペア (BRP) の内容に依存します。
- ウォッチポイントデバッグ イベントは、関連するウォッチポイントレジスタ ペア (WRP) の内容に依存します。
- リンク付きブレークポイントまたはウォッチポイントデバッグ イベントは、2 番目の BRP の設定に依存します。
- ベクタキャッチデバッグ イベントは、ベクタキャッチ レジスタ (DBGVCR) の設定に依存します。
- OS ロック解除キャッチデバッグ イベントは、イベントキャッチ レジスタ (DBGECR) の設定に依存します。

これらに加えて、P. C3-2 の表 C3-1 に示されているように、デバッグイベントの生成は次の設定にも依存します。

- 侵襲性デバッグの認証の設定。第 C2 章*「侵襲性デバッグの認証」*を参照して下さい。
- DBGDSCR.HDBGen および DBGDSCR.MDBGen ビットの値。*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。

次に示すイベントは、次の ISB 操作、例外開始、または例外からの復帰の終了時に、デバッグイベント生成ロジックにより必ず実行されます。

- 次のものを含むコンテキスト変更操作
  - – モード変更
  - – CONTEXTIDR への書き込み
  - – セキュリティ状態の変更
- 次のものを含む、デバッグイベント生成ロジックの状態を変更する操作
  - – ブレークポイントデバッグ イベント、あるいはリンク付きブレークポイントまたはウォッチポイントデバッグ イベントの BRP レジスタへの書き込み
  - – ウォッチポイントデバッグ イベントの WRP レジスタへの書き込み
  - – ベクタキャッチデバッグ イベントの DBGVCR への書き込み
  - – OS ロック解除キャッチデバッグ イベントの DBGECR への書き込み
  - – 認証信号の変化
  - – DBGDSCR への書き込み

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Usually, exception return sequences are also context changing operations, and hence the context change operation is guaranteed to take effect on the breakpoint matching logic by the end of that exception return sequence.

To ensure a change in the debug event generation logic has completed before a particular event or piece of code is debugged you must include an ISB, exception entry or exception return after the change in the Debug settings. In the absence of an ISB, exception entry or exception return, it is UNPREDICTABLE when the changes take place.

Between a context change operation and the end of the next ISB, exception entry or exception return it is UNPREDICTABLE whether the processing of a debug event depends on the old or the new context.

Between operations that change the state of the debug event generation logic and the end of the next ISB, exception entry or exception return, it is UNPREDICTABLE whether debug event generation depends on the old or the new settings. Example C3-1 describes such a case.

Example C3-1 Unpredictability in debug event generation

A breakpoint is set at an address programmed in its Breakpoint Value Register (DBGBVR) and is configured through its Breakpoint Control Register (DBGBCR). In this example:

- DBGBCR is programmed to only match in User, Supervisor or System modes
- the address in the DBGBVR is the address of an instruction in an abort handler routine normally entered from the Prefetch Abort exception vector in Abort mode, but located after that handler switches from Abort mode to Supervisor mode using a CPS instruction.

If there is no ISB, exception entry or exception return between the CPS instruction and the instruction at the breakpoint address, it is UNPREDICTABLE whether the breakpoint matches, even though the instruction is executed in Supervisor mode.

Such an ISB, exception entry or exception return is usually not required to ensure correct operation of the program. In this example because the program is switching between two privileged modes it is not required to ensure correct operation of the memory system.

ARMv7 does not require that such changes take effect on instruction fetches from the memory system, or on memory accesses made by the processor, at the same point as they take effect on the debug logic. The only architectural requirement is that such a change executed before an ISB operation must be visible to both the memory system and the debug logic for all instructions executed after the ISB operation. This requirement is described earlier in this section.

The processor must test for any possible:

- Watchpoint debug event before a memory access operation is observed.
- Breakpoint or Vector Catch debug event before the instruction is executed, that is, before the instruction has any effect on the architectural state of the processor.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

通常、例外からの復帰シーケンスはコンテキスト変更操作でもあり、この理由からコンテキスト変更操作は、その例外からの復帰シーケンスの終了時に、ブレークポイント一致ロジックに必ず適用されます。

デバッグイベント生成ロジックの変更が、特定のイベントまたはコードの一部がデバッグされるまえに完了することを保証するため、デバッグ設定の変更後には ISB、例外開始、例外からの復帰のいずれかを含める必要があります。ISB、例外開始、または例外からの復帰が存在しない場合、変更がどの時点で適用されるかは予測不能です。

コンテキスト変更操作と、次の ISB、例外開始、または例外からの復帰との間で、デバッグイベントの処理が古いコンテキストと新しいコンテキストのどちらに従って実行されるかは予測不能です。

デバッグイベント生成ロジックの状態を変更する操作と、次の ISB、例外開始、または例外からの復帰との間で、デバッグイベントの生成が古い設定と新しい設定のどちらに従って行われるかは予測不能です。このような状況について、例 C3-1 に示します。

例 C3-1 デバッグイベント生成での予測不能性

ブレークポイントは、ブレークポイント値レジスタ (DBGBVR) でプログラムされたアドレスに設定され、ブレークポイント制御レジスタ (DBGBCR) 経由で構成されます。この例では、次の構成を想定します。

- DBGBCR は、ユーザ、スーパバイザ、システムモードでのみ一致するようにプログラムされます。
- DBGBVR のアドレスは、アボートモードでのプリフェッチアボート例外ベクタから通常開始されるアボートハンドラ ルーチンの命令のアドレスですが、ハンドラが CPS 命令を使用してアボートモードからスーパバイザモードに切り替わった後に置かれています。

CPS 命令と、ブレークポイント アドレスにある命令との間に、ISB、例外開始、または例外からの復帰が存在しない場合、ブレークポイントが一致するかどうかは、命令がスーパバイザモードで実行されている場合も含めて予測不能です。

このような ISB、例外開始、または例外からの復帰は、プログラムの正しい動作を保証するために通常は必要ありません。この例では、プログラムが 2 つの特権モードの間で切り替わるため、メモリシステムの正しい動作を保証する必要はありません。

ARMv7 では、このような変更がメモリシステムからの命令フェッチや、プロセッサによって行われるメモリアクセスに対して、デバッグロジックに対してと同時に適用されることが要求されてはいません。アーキテクチャで要求されるのは、ISB 操作の前にこのような変更が実行された場合、ISB 操作の後で実行されるすべての命令について、メモリシステムとデバッグロジックの両方に対してその変更が可視であることのみです。この条件については、このセクションですでに説明しています。

プロセッサは、次に示す可能性をすべてテストする必要があります。

- メモリアクセス操作が観測される前のウォッチポイントデバッグ イベント

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

As a result, for an instruction that modifies the context in which the processor tests for debug events, the processor must test for all possible debug event in terms of the context before the memory access operation is observed or the instruction executes. For example:

- In a v7 Debug implementation that uses the memory-mapped interface, a write to the DBGWCR to enable a watchpoint on a Data Virtual Address (DVA) of the DBGWCR itself must not trigger the watchpoint.

  Conversely, a write to the DBGWCR to disable the same watchpoint must trigger the watchpoint. For more information, see *Debug events in the debug monitor* on page C3-26.

- An instruction that writes to a Breakpoint Control Register (DBGBCR) or Vector Catch Register (DBGVCR) to enable a debug event on the Instruction Virtual Address (IVA) of the instruction itself *must not* trigger the debug event.

  Conversely, a write to the DBGBCR or DBGVCR to disable the same debug event must trigger the debug event.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- 命令が実行される前、つまりその命令がプロセッサのアーキテクチャ的な状態に何らかの影響を及ぼす前の、ブレークポイントまたはベクタキャッチデバッグ イベント

その結果、プロセッサがデバッグイベントをテストするコンテキストを変更する命令について、メモリアクセス操作が観測される、または命令が実行される前に、プロセッサはコンテキストに関して可能なデバッグイベントのすべてをテストする必要があります。次に例を示します。

- メモリマップされたインタフェースを使用する v7 デバッグ実装で、DBGWCR への書き込みにより、その DBGWCR 自身のデータ仮想アドレス (DVA) へのウォッチポイントが可能になる場合、その書き込み自体がウォッチポイントをトリガすることはできません。

  逆に、DBGWCR への書き込みが同じウォッチポイントを不可能にする場合、その書き込みは必ずウォッチポイントをトリガしなければなりません。詳細については、*デバッグモニタでのデバッグイベント* P. C3-26 を参照して下さい。

- ブレークポイント制御レジスタ (DBGBCR) またはベクタキャッチ レジスタ (DBGVCR) へ書き込みを行う命令により、その命令自体の命令仮想アドレス (IVA) に対してデバッグイベントが可能になる場合、その書き込みがデバッグイベントをトリガすることは *できません*。

  逆に、DBGBCR または DBGVCR への書き込みによって同じデバッグイベントが不可能になる場合、その書き込みは必ずデバッグイベントをトリガしなければなりません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C3.5 Debug event prioritization

Debug events can be synchronous or asynchronous:

- Breakpoint, Vector Catch, BKPT Instruction, and synchronous Watchpoint debug events are all synchronous debug events
- asynchronous Watchpoints and all Halting debug events are all asynchronous debug events.

A single instruction can generate a number of synchronous debug events. It can also generate a number of synchronous exceptions. The principles given in *Exception priority order* on page B1-33 apply to those exceptions and debug events, in addition to the following:

- An instruction fetch that generates an MMU fault, MPU fault, or external abort does not generate a Breakpoint or Vector Catch debug event.
- Breakpoint and Vector Catch debug events are associated with the instruction and are taken before the instruction executes. Therefore, when a Breakpoint or Vector Catch debug event occurs no other synchronous exception or debug event that would have occurred as a result of executing the instruction is generated.
- If a single instruction has more than one of the following debug events associated with it, it is UNPREDICTABLE which is taken:
  - Breakpoint
  - Vector Catch.
- No instruction is valid if it has a Prefetch Abort exception associated with it. Therefore, if an instruction causes a Prefetch Abort exception no other synchronous exception or debug event that would have occurred as a result of executing the instruction is generated.
- An instruction that generates an Undefined Instruction exception does not cause any memory access, and therefore cannot cause a Data Abort exception or a Watchpoint debug event.
- A memory access that generates an MMU fault or an MPU fault must not generate a Watchpoint debug event.
- A memory access that generates an MMU fault, an MPU fault, or a synchronous Watchpoint debug event must not generate an external abort.
- All other synchronous exceptions and synchronous debug events are mutually exclusive, and are derived from a decode of the instruction.

The ARM architecture does not define when asynchronous debug events other than asynchronous Watchpoint debug events are taken. Therefore the prioritization of asynchronous debug events other than asynchronous Watchpoint debug events is IMPLEMENTATION DEFINED.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C3.5 デバッグイベントの優先度付け

デバッグイベントは、同期または非同期です。

- ブレークポイント、ベクタキャッチ、BKPT 命令、同期ウォッチポイントデバッグ イベントは、すべて同期デバッグイベントです。
- 非同期ウォッチポイントおよびすべてのホールトデバッグ イベントは非同期デバッグイベントです。

単一の命令で、多くの同期デバッグイベントを生成することができます。また、多くの同期例外を生成することもできます。これらの例外およびデバッグイベントに対しては、*例外の優先順位* P. B1-33 に示されている原則に加えて、次に示す原則が適用されます。

- MMU フォールト、MPU フォールト、または外部アボートを生成する命令フェッチは、ブレークポイントまたはベクタキャッチデバッグ イベントを生成しません。
- ブレークポイントおよびベクタキャッチデバッグ イベントは、命令と関連付けられ、その命令の実行前に取得されます。このため、ブレークポイントまたはベクタキャッチデバッグ イベントが発生するとき、その命令の実行結果として発生するべき、他の同期例外やデバッグイベントは生成されません。
- 単一の命令に、次に示すデバッグイベントの 2 つ以上が関連付けられている場合、どれが取得されるかは予測不能です。
  - ブレークポイント
  - ベクタキャッチ
- プリフェッチアボート例外が関連付けられた命令は無効です。このため、命令でプリフェッチアボート例外が発生する場合、その命令の実行結果として発生するべき、他の同期例外やデバッグイベントは生成されません。
- 未定義命令例外を生成する命令は、メモリアクセスを引き起こさないため、データアボート例外またはウォッチポイントデバッグ イベントを引き起こすことができません。
- MMU フォールトまたは MPU フォールトを生成するメモリアクセスは、ウォッチポイントデバッグ イベントを生成できません。
- MMU フォールト、MPU フォールト、または同期ウォッチポイントデバッグ イベントを生成するメモリアクセスは、外部アボートを生成できません。
- 他のすべての同期例外および同期デバッグイベントは相互排他的で、命令のデコードから派生します。

ARM アーキテクチャでは、非同期ウォッチポイントデバッグ イベントを除く非同期デバッグイベントが、どの時点で取得されるかは定義されていません。このため、非同期ウォッチポイントデバッグ イベントを除く、非同期デバッグイベントの優先度付けは実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Debug events must be taken in the execution order of the sequential execution model. This means that if an instruction causes a debug event then that event must be taken before any debug event on any instruction that would execute after that instruction, in the sequential execution model. In particular, if the execution of an instruction generates an asynchronous Watchpoint debug event:

- the asynchronous Watchpoint debug event must not be taken if the instruction also generates any synchronous debug event
- if the instruction does not generate any synchronous debug event, then the asynchronous Watchpoint debug event must be taken before any subsequent:
  - synchronous or asynchronous debug event
  - synchronous or asynchronous precise exception.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

デバッグイベントは、シーケンシャル実行モデルの実行順序で取得される必要があります。つまり、命令がデバッグイベントを引き起こす場合、そのイベントは、シーケンシャル実行モデルにおいて、その命令よりも後に実行されるどの命令に対するデバッグイベントよりも前に取得される必要があります。特に、命令の実行により非同期ウォッチポイントデバッグ イベントが生成される場合、次の条件が適用されます。

- その命令が同期デバッグイベントも生成する場合、その非同期ウォッチポイントデバッグ イベントを取得することはできません。
- その命令が同期デバッグイベントを生成しない場合、その非同期ウォッチポイントデバッグ イベントは、以後の次のようなイベントよりも前に取得される必要があります。
  - 同期または非同期デバッグイベント
  - 同期または非同期の正確な例外

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter C4
Debug Exceptions

This chapter describes debug exceptions, that are used to handle debug events when the processor is configured for Monitor debug-mode. It contains the following sections:

- *About debug exceptions* on page C4-2
- *Effects of debug exceptions on CP15 registers and the DBGWFAR* on page C4-4.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 C4 章
デバッグ例外

本章ではデバッグ例外について説明します。デバッグ例外は、プロセッサがモニタデバッグモード用に構成されているときのデバッグイベントを処理するために使用されます。本章は次のセクションから構成されています。

- *デバッグ例外について* P. C4-2
- *CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響* P. C4-4

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C4.1 About debug exceptions

A debug exception is taken when:

- a permitted Software debug event occurs when invasive debug is enabled and Monitor debug-mode is selected
- a BKPT instruction is executed when one of:
  - — invasive debug is disabled
  - — the debug event is not permitted
  - — no debug-mode is selected.

For more information, see Table C3-1 on page C3-2. You must be careful when programming certain events because you might leave the processor in an unrecoverable state. See *Unpredictable behavior on Software debug events* on page C3-24.

How the processor handles the debug exception depends on the cause of the exception, and is described in:

- *Debug exception on Breakpoint, BKPT Instruction or Vector Catch debug events*
- *Debug exception on Watchpoint debug event* on page C4-3.

Halting debug events never cause a debug exception. The Halting debug events are:

- External Debug Request debug event
- Halt Request debug event
- OS Unlock Catch debug event.

C4.1.1 Debug exception on Breakpoint, BKPT Instruction or Vector Catch debug events

If the cause of the debug exception is a Breakpoint, BKPT Instruction, or a Vector Catch debug event, the processor performs the following actions:

- Sets the DBGDSCR.MOE bits according to Table C10-3 on page C10-26.
- Sets the IFSR and IFAR as described in *Effects of debug exceptions on CP15 registers and the DBGWFAR* on page C4-4.
- Generates a Prefetch Abort exception, see *Prefetch Abort exception* on page B1-54

The Prefetch Abort handler is responsible for checking the IFSR bits to find out whether the exception entry was caused by a debug exception. If it was, typically the handler branches to the debug monitor.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C4.1 デバッグ例外について

デバッグ例外は、次の場合に取得されます。

- 侵襲性デバッグが可能で、モニタデバッグ モードが選択されているときに、許可されているソフトウェアデバッグ イベントが発生した。
- 次のいずれかの条件で、BKPT 命令が実行された。
  - – 侵襲性デバッグが不可能である。
  - – デバッグイベントが許可されていない。
  - – デバッグモードが選択されていない。

詳細については、P. C3-2 の表 C3-1 を参照して下さい。特定のイベントではプロセッサが回復不能な状態に陥ることがあるため、これらのイベントをプログラムするときは注意する必要があります。*ソフトウェアデバッグ イベントでの予測不能な動作* P. C3-24 を参照して下さい。

プロセッサがデバッグ例外をどのように処理するかは、例外の原因によって異なります。詳細については、次に示すセクションを参照して下さい。

- *ブレークポイント、BKPT 命令、ベクタキャッチデバッグ イベントでのデバッグ例外*
- *ウォッチポイントデバッグ イベントでのデバッグ例外* P. C4-3

ホールトデバッグ イベントがデバッグ例外を引き起こすことはありません。ホールトデバッグ イベントは次のいずれかです。

- 外部デバッグ要求デバッグイベント
- ホールト要求デバッグイベント
- OS ロック解除キャッチデバッグ イベント

C4.1.1 ブレークポイント、BKPT 命令、ベクタキャッチデバッグ イベントでのデバッグ例外

デバッグ例外の原因がブレークポイント、BKPT 命令、ベクタキャッチデバッグ イベントのいずれかの場合、プロセッサは次の動作を実行します。

- P. C10-26 の表 C10-3 に示すように DBGDSCR.MOE ビットを設定します。
- *CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響* P. C4-4 に示すように IFSR および IFAR を設定します。
- プリフェッチアボート例外を生成します。*プリフェッチアボート例外* P. B1-54 を参照して下さい。

プリフェッチアボート ハンドラは、IFSR ビットをチェックし、例外開始がデバッグ例外によって引き起こされたものかどうかを判別します。そうである場合、ハンドラは通常はデバッグモニタに分岐します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C4.1.2 Debug exception on Watchpoint debug event

If the cause of the debug exception is a Watchpoint debug event, the processor performs the following actions:

- Sets the DBGDSCR.MOE bits either to Asynchronous Watchpoint Occurred or to Synchronous Watchpoint Occurred.
- Sets the DFSR, DFAR, and DBGWFAR as described in *Effects of debug exceptions on CP15 registers and the DBGWFAR* on page C4-4.
- Generates a precise Data Abort exception, see *Data Abort exception* on page B1-55.

  For more information, see *Synchronous and Asynchronous Watchpoint debug events* on page C3-18.

The Data Abort handler is responsible for checking the DFSR bits to find out whether the exception entry was caused by a debug exception. If it was, typically the handler branches to the debug monitor:

- The DBGWFAR indicates the address of the instruction that caused the Watchpoint debug event. see *Watchpoint Fault Address Register (DBGWFAR)* on page C10-28.
- LR_abt holds the address of (instruction to restart at + 8). If the watchpoint is synchronous, the instruction to restart at is the instruction that triggered the watchpoint.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C4.1.2 ウォッチポイントデバッグ イベントでのデバッグ例外

デバッグ例外の原因がウォッチポイントデバッグ イベントの場合、プロセッサは次の動作を実行します。

- DBGDSCR.MOE ビットを、「非同期ウォッチポイントの発生」または「同期ウォッチポイントの発生」にセットします。
- DFSR、DFAR、DBGWFAR を、*CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響* P. C4-4 に示されているように設定します。
- 正確データアボート例外を生成します。*データアボート例外* P. B1-55 を参照して下さい。

  詳細については、*同期と非同期のウォッチポイントデバッグ イベント* P. C3-18 を参照して下さい。

データアボート ハンドラは、DFSR ビットをチェックし、例外開始がデバッグ例外によって引き起こされたものかどうかを判別します。そうである場合、ハンドラは通常はデバッグモニタに分岐します。

- DBGWFAR は、ウォッチポイントデバッグ イベントを引き起こした命令のアドレスを示しています。*ウォッチポイント フォールト アドレス レジスタ (DBGWFAR)* P. C10-28 を参照して下さい。
- LR_abt には、(8 バイト先の再開する命令) のアドレスが保持されています。ウォッチポイントが同期の場合、再開する命令は、ウォッチポイントをトリガした命令です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C4.2 Effects of debug exceptions on CP15 registers and the DBGWFAR

There are four CP15 registers that are used to record abort information:

DFAR Data Fault Address Register, see:

- *c6, Data Fault Address Register (DFAR)* on page B3-124 for a VMSA implementation
- *c6, Data Fault Address Register (DFAR)* on page B4-57 for a PMSA implementation.

IFAR Instruction Fault Address Register, see:

- *c6, Instruction Fault Address Register (IFAR)* on page B3-125 for a VMSA implementation
- *c6, Instruction Fault Address Register (IFAR)* on page B4-58 for a PMSA implementation.

DFSR Data Fault Status Register, see:

- *c5, Data Fault Status Register (DFSR)* on page B3-121 for a VMSA implementation
- *c5, Data Fault Status Register (DFSR)* on page B4-55 for a PMSA implementation.

IFSR Instruction Fault Status Register, see:

- *c5, Instruction Fault Status Register (IFSR)* on page B3-122 for a VMSA implementation
- *c5, Instruction Fault Status Register (IFSR)* on page B4-56 for a PMSA implementation.

Their usage model for normal operation is described in:

- *Fault Status and Fault Address registers in a VMSA implementation* on page B3-48 for a VMSA implementation
- *Fault Status and Fault Address registers in a PMSA implementation* on page B4-18 for a PMSA implementation.

Additional registers might be used to return additional IMPLEMENTATION DEFINED fault status information, see:

- *c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B3-123 for a VMSA implementation
- *c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B4-56 for a PMSA implementation.

Also, information can be returned in the *Watchpoint Fault Address Register* (DBGWFAR). The implementation of the DBGWFAR depends on the Debug architecture version:

- In v6 Debug it is implemented as a register in CP15 c6.
- In v6.1 Debug it is implemented in CP14, and use of the CP15 alias is deprecated.
- In v7 Debug it can be implemented in the Extended CP14 interface, and has no alias in CP15.

For more information, see *Watchpoint Fault Address Register (DBGWFAR)* on page C10-28.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C4.2 CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響

アボート情報を記録するために使用される CP15 レジスタは 4 つあり、それぞれ次の情報を保持します。

DFAR データフォールトアドレス レジスタ。詳細については、次に示すセクションを参照して下さい。

- *c6、データフォールト アドレスレジスタ (DFAR)* P. B3-124（VMSA 実装の場合）
- *c6、データフォールト アドレスレジスタ (DFAR)* P. B4-57 （PMSA 実装の場合）

IFAR 命令フォールトアドレス レジスタ。詳細については、次に示すセクションを参照して下さい。

- *c6、命令フォールト アドレスレジスタ (IFAR)* P. B3-125（VMSA 実装の場合）
- *c6、命令フォールト アドレスレジスタ (IFAR)* P. B4-58 （PMSA 実装の場合）

DFSR データフォールトステータス レジスタ。詳細については、次に示すセクションを参照して下さい。

- *c5、データフォールト ステータスレジスタ (DFSR)* P. B3-121（VMSA 実装の場合）
- *c5、データフォールト ステータスレジスタ (DFSR)* P. B4-55 （PMSA 実装の場合）

IFSR 命令フォールトステータス レジスタ。詳細については、次に示すセクションを参照して下さい。

- *c5、命令フォールト ステータスレジスタ (IFSR)* P. B3-122（VMSA 実装の場合）
- *c5、命令フォールト ステータスレジスタ (IFSR)* P. B4-56 （PMSA 実装の場合）

通常の動作におけるこれらの使用モデルについては、次に示すセクションを参照して下さい。

- *VMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B3-48（VMSA 実装の場合）
- *PMSA 実装のフォールトステータス レジスタとフォールトアドレス レジスタ* P. B4-18（PMSA 実装の場合）

実装定義の追加フォールトステータス情報を返すため、他のレジスタが使用されることがあります。詳細については、次に示すセクションを参照して下さい。

- *c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B3-123（VMSA 実装の場合）
- *c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B4-56 （PMSA 実装の場合）

ウォッチポイントフォールトアドレス レジスタ(DBGWFAR) から情報を返すこともできます。DBGWFAR の実装は、デバッグアーキテクチャのバージョンによって異なります。

- v6 デバッグでは、CP15 c6 のレジスタとして実装されています。
- v6.1 デバッグでは CP14 に実装されており、CP15 エイリアスの使用は推奨されません。
- v7 デバックでは拡張 CP14 インタフェースに実装可能で、CP15 のエイリアスは存在しません。

詳細については、*ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10-28 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In Monitor debug-mode the behavior on the exception generated as a result of a Breakpoint, BKPT Instruction, or Vector Catch debug events is as follows:

- the IFSR is updated with the encoding for a debug event, IFSR[10,3:0] = 0b00010
- the IFAR is UNKNOWN following these debug exceptions
- the DFSR, DFAR and DBGWFAR are unchanged.

In Monitor debug-mode the behavior on the exception generated as a result of a Watchpoint debug event is as follows:

- the IFSR and IFAR are unchanged.
- the DFSR is updated with the encoding for a debug event, DFSR[10,3:0] = 0b00010.
- the Domain and Write fields in the DFSR, DFSR[11,7:4], are UNKNOWN. However, an ARMv6 watchpoint sets the Domain field.
- the DFAR is UNKNOWN.
- the DBGWFAR is updated with the Instruction Virtual Address (IVA) of the instruction that accessed the watchpointed address, plus an offset that depends on the instruction set state of the processor for that instruction:
  - 8 in ARM state
  - 4 in Thumb and ThumbEE states
  - IMPLEMENTATION DEFINED in Jazelle state.

  See *Memory addresses* on page C3-23 for a definition of the IVA used to update the DBGWFAR.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

モニタデバッグ モードでは、ブレークポイント、BKPT 命令、ベクタキャッチ デバッグイベントの結果として生成された例外時の動作は次のとおりです。

- デバッグイベントのエンコードで IFSR が更新され、IFSR[10、3:0] = 0b00010 になります。
- これらのデバッグ例外が発生した後の IFAR の値は未知です。
- DFSR、DFAR、DBGWFAR は変更されません。

モニタデバッグ モードでは、ウォッチポイントデバッグ イベントの結果として生成された例外時の動作は次のとおりです。

- IFSR および IFAR は変更されません。
- デバッグイベントのエンコードで DFSR が更新され、DFSR[10、3:0] = 0b00010 になります。
- DFSR の Domain および Write フィールドである DFSR[11、7:4] は未知です。ただし、ARMv6 ウォッチポイントでは Domain フィールドが設定されます。
- DFAR の値は未知です。
- DBGWFAR は、ウォッチポイントが設定されたアドレスにアクセスした命令の命令仮想アドレス (IVA) に、その命令に対するプロセッサの命令セット状態によって異なるオフセットを加えた値で更新されます。
  - ARM 状態では 8
  - Thumb および ThumbEE 状態では 4
  - Jazelle 状態では実装定義

  DBGWFAR の更新に使用される IVA の定義については、*メモリアドレス* P. C3- 23 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter C5
Debug State

This chapter describes Debug state, that is entered if a debug event occurs when the processor is configured for Halting debug-mode. It contains the following sections:

- *About Debug state* on page C5-2
- *Entering Debug state* on page C5-3
- *Behavior of the PC and CPSR in Debug state* on page C5-7
- *Executing instructions in Debug state* on page C5-9
- *Privilege in Debug state* on page C5-13
- *Behavior of non-invasive debug in Debug state* on page C5-19
- *Exceptions in Debug state* on page C5-20
- *Memory system behavior in Debug state* on page C5-24
- *Leaving Debug state* on page C5-28.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 C5 章
デバッグ状態

本章では、デバッグ状態について説明します。プロセッサがホールトデバッグ モードに構成されているときデバッグイベントが発生すると、デバッグ状態が開始されます。本章は次のセクションから構成されています。

- *デバッグ状態について* P. C5-2
- *デバッグ状態への遷移* P. C5-3
- *デバッグ状態での PC および CPSR の動作* P. C5-7
- *デバッグ状態での命令の実行* P. C5-9
- *デバッグ状態での特権* P. C5-13
- *デバッグ状態での非侵襲性デバッグの動作* P. C5-19
- *デバッグ状態での例外* P. C5-20
- *デバッグ状態でのメモリシステムの動作* P. C5-24
- *デバッグ状態の終了* P. C5-28

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.1 About Debug state

When invasive debug is enabled, the processor switches to a special state called Debug state if one of:

- a permitted Software debug event occurs and Halting debug-mode is selected
- a permitted Halting debug event occurs
- a Halting debug event becomes permitted while it is pending.

For more information, see *State* on page B1-3. In Debug state, control passes to an external agent.

Note

The external agent is usually a debugger. However it might be some other agent connecting to the debug port of the processor. This could be another processor in the same *System on Chip* (SoC) device. In part C of this manual this agent is often referred to as a debugger.

In v6 Debug, when debug is enabled and Halting debug-mode is not selected it is IMPLEMENTATION DEFINED whether a Halting debug event causes entry to Debug state. For more information, see Table C3-1 on page C3-2.

Halting debug-mode is configured by setting DBGDSCR[14] to 1, see *Debug Status and Control Register (DBGDSCR)* on page C10-10.

Parts A and B of this manual describe how an ARMv7 processor behaves when it is not in Debug state, that is, when it is in Non-debug state. In Debug state, the processor behavior changes as follows:

- The PC and CPSR behave as described in *Behavior of the PC and CPSR in Debug state* on page C5-7.
- Instructions are prefetched from the Instruction Transfer Register (DBGITR), see *Executing instructions in Debug state* on page C5-9.
- The processor can execute only instructions from the ARM instruction set.
- The rules about modes and privileges are different to those in Non-debug state, see *Privilege in Debug state* on page C5-13.
- Non-invasive debug features are disabled, see *Behavior of non-invasive debug in Debug state* on page C5-19.
- Exceptions are treated as described in *Exceptions in Debug state* on page C5-20. Other software and Halting debug events and interrupts are ignored.
- If the processor implements a DMA engine, its behavior is IMPLEMENTATION DEFINED.
- If the processor implements a cache or other local memory that it keeps coherent with other memories in the system during normal operation, it must continue to service coherency requests from the other memories.

Leaving Debug state on page C5-28 describes how to leave Debug state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C5.1 デバッグ状態について

侵襲性デバッグが可能なとき、次の条件のいずれかが満たされると、プロセッサはデバッグ状態と呼ばれる特別な状態に切り替わります。

- 許可されているソフトウェアデバッグ イベントが発生し、ホールトデバッグ モードが選択されている。
- 許可されているホールトデバッグ イベントが発生した。
- ホールトデバッグ イベントが保留中のときに許可された。

詳細については、*状態* P. B1-3 を参照して下さい。デバッグ状態では、コントロールは外部のエージェントに渡されます。

注

外部のエージェントは、通常はデバッガです。ただし、プロセッサのデバッグポートに接続している他のエージェントの場合もあります。これは、同じ *システム オンチップ*(SoC) デバイスに存在する他のプロセッサの場合もあります。本書のパート C では、多くの場合このエージェントをデバッガと呼びます。

v6 デバッグでは、デバッグが可能でホールトデバッグ モードが選択されていない場合、ホールトデバッグ イベントによりデバッグ状態に遷移するかどうかは実装定義です。詳細については、P. C3-2 の表 C3-1 を参照して下さい。

ホールトデバッグ モードは、DBGDSCR[14] を 1 にセットすることで構成されます。*デバッグ ステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。

本書のパート A とパート B では、ARMv7 プロセッサがデバッグ状態でない、つまり非デバッグ状態のときにどのように動作するかについて説明しています。デバッグ状態では、プロセッサの動作は次のように変化します。

- PC と CPSR は、*デバッグ状態での PC および CPSR の動作* P. C5-7 で説明されているように動作します。
- 命令は、命令転送レジスタ (DBGITR) からプリフェッチされます。*デバッグ状態での命令の実行* P. C5-9 を参照して下さい。
- プロセッサは、ARM 命令セットからの命令のみを実行できます。
- モードと特権についての規則は、非デバッグ状態のものとは異なります。*デバッグ状態での特権* P. C5-13 を参照して下さい。
- 非侵襲性デバッグ機能は不可能です。*デバッグ状態での非侵襲性デバッグの動作* P. C5-19 を参照して下さい。
- 例外は、*デバッグ状態での例外* P. C5-20 で説明されているように取り扱われます。他のソフトウェアおよびホールトデバッグ イベント、および割り込みは無視されます。
- プロセッサに DMA エンジンが実装されている場合、その動作は実装定義です。
- プロセッサにキャッシュまたは他のローカルメモリが実装されており、通常動作においてシステムの他のメモリとコヒーレンシを維持するようになっている場合、他のメモリからのコヒーレンシ要求に対してサービスを続ける必要があります。

デバッグ状態の終了方法については、*デバッグ状態の終了* P. C5-28 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.2 Entering Debug state

When invasive debug is enabled, the processor switches to a special state called Debug state if one of:

- a permitted Software debug event occurs and Halting debug-mode is selected
- a permitted Halting debug event occurs
- a Halting debug event becomes permitted while it is pending.

In v6 Debug, when debug is enabled and Halting debug-mode is not selected it is IMPLEMENTATION DEFINED whether a Halting debug event causes entry to Debug state. For more information, see Table C3-1 on page C3-2.

Note

Entering Debug state does not ensure that the effect of any context altering operation performed before Debug state entry is visible to instructions executed in Debug state.

On entering Debug state the processor follows this sequence:

1. The processor signals to the system that it is entering Debug state. Details of the signalling method, including whether it is implemented, are IMPLEMENTATION DEFINED.
2. Processing is halted, meaning:
   - The instruction pipeline is flushed and no more instructions are prefetched from memory.
   - The values of the following are not changed on entering Debug state:
     - the PC and CPSR
     - all general-purpose and program status registers, including SPSR_abt and LR_abt.
   - The values of the PC and CPSR remain unchanged while the processor is in Debug state.
   - Instructions can be executed in Debug state, see *Executing instructions in Debug state* on page C5-9, but when the instruction is executed in this way the normal effects of incrementing the PC and updating the CPSR are masked.
   - The effect of Debug state entry on CP15 registers and debug registers is described in *Effect of entering Debug state on CP15 registers and the DBGWFAR* on page C5-4.
   - The processor signals to the system that it is in Debug state. Details of this signalling method, including whether it is implemented, are IMPLEMENTATION DEFINED.
   - The processor might:
     - ensure that all Non-debug state memory operations complete and signal this to the system
     - set the DBGDSCR.ADAdiscard bit to 1.

     However, processor behavior regarding memory accesses outstanding at Debug state entry is IMPLEMENTATION DEFINED, see *Asynchronous aborts and entry to Debug state* on page C5-5.

     Details of the method used to signal to the system that Non-debug state memory operations are complete, including whether any such method is implemented, are IMPLEMENTATION DEFINED.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C5.2 デバッグ状態への遷移

侵襲性デバッグが可能なとき、次の条件のいずれかが満たされると、プロセッサはデバッグ状態と呼ばれる特別な状態に切り替わります。

- 許可されているソフトウェアデバッグ イベントが発生し、ホールトデバッグ モードが選択されている。
- 許可されているホールトデバッグ イベントが発生した。
- ホールトデバッグ イベントが保留中のときに許可された。

v6 デバッグでは、デバッグが可能でホールトデバッグ モードが選択されていない場合、ホールトデバッグ イベントによりデバッグ状態に遷移するかどうかは実装定義です。詳細については、P. C3-2 の表 C3-1 を参照して下さい。

注

デバッグ状態への遷移では、状態遷移の前に実行されたコンテキスト変更操作のいずれかの影響が、デバッグ状態で実行される命令から可視であることは保証されません。

デバッグ状態への遷移時に、プロセッサは次のシーケンスを実行します。

1. プロセッサはシステムに対して、デバッグ状態に遷移することを通知する信号を送信します。信号の送信方法の詳細は、それを実装するかどうかも含めて実装定義です。
2. プロセッサがホールトします。これは次のことを意味します。
   - 命令パイプラインがフラッシュされ、新しい命令がメモリからプリフェッチされなくなります。
   - デバッグ状態への遷移時に、次のレジスタの値は変更されません。
     - PC および CPSR
     - SPSR_abt および LR_abt を含む、すべての汎用およびプログラムステータスレジスタ
   - PC および CPSR の値は、プロセッサがデバッグ状態の間は変更されません。
   - デバッグ状態で命令を実行することはできます。*デバッグ状態での命令の実行* P. C5-9 を参照して下さい。ただし、この方法で命令が実行されるときは、通常の効果である、PC のインクリメントと CPSR の更新はマスクされます。
   - デバッグ状態への遷移による CP15 レジスタおよびデバッグレジスタへの影響については、*デバッグ状態への遷移が CP15 レジスタおよび DBGWFAR に及ぼす影響* P. C5-4 を参照して下さい。
   - プロセッサはシステムに対して、デバッグ状態であることを通知する信号を送信します。信号の送信方法の詳細は、それを実装するかどうかも含めて実装定義です。
   - プロセッサは次の動作を行うことがあります。
     - 非デバッグ状態のメモリ操作すべてが完了したことを保証し、そのことをシステムに通知します。
     - DBGDSCR.ADAdiscard ビットを 1 にセットします。

     ただし、デバッグ状態への遷移時に未解決であるメモリアクセスに関するプロセッサの動作は実装定義です。*非同期アボートとデバッグ状態への遷移* P. C5-5 を参照して下さい。

     非デバッグ状態のメモリ操作が完了したことをシステムに通知するため使用される手法の詳細は、そのような手法を実装するかどうかも含めて、実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

3. The processor signals that it has entered Debug state and is ready for an external agent to take control:
   - the DBGDSCR.HALTED bit is set to 1
   - the DBGDSCR.MOE field is set according to Table C10-3 on page C10-26.

For details of the recommended external debug interface, see *Run-control and cross-triggering signals* on page AppxA-5 and *DBGACK and DBGCPUDONE* on page AppxA-7.

C5.2.1 Effect of entering Debug state on CP15 registers and the DBGWFAR

The actions taken on entering Debug state depend on what caused the Debug state entry:

- If Debug state was entered following a Watchpoint debug event, then the DBGWFAR is updated with the Instruction Virtual Address (IVA) of the instruction that accessed the watchpointed address, plus an offset that depends on the instruction set state of the processor when the debug event was generated:
  - — 8 in ARM state
  - — 4 in Thumb and ThumbEE states
  - — IMPLEMENTATION DEFINED in Jazelle state.

  See *Memory addresses* on page C3-23 for a definition of the IVA used to update the DBGWFAR.
- Otherwise, the DBGWFAR is unchanged on entry to Debug state.

Note

- The implementation of the DBGWFAR depends on the Debug architecture version:
  - — In v6 Debug it is implemented as a register in CP15 c6.
  - — In v6.1 Debug it is implemented in CP14, and use of the CP15 alias is deprecated.
  - — In v7 Debug it can be implemented in the Extended CP14 interface, and has no alias in CP15.

  For more information, see *Watchpoint Fault Address Register (DBGWFAR)* on page C10-28.
- In all cases, on Debug state entry the DBGWFAR is set as described in this section.

In ARMv7, all CP15 registers are unchanged on entry to Debug state. In ARMv6, all CP15 registers except for the DBGWFAR are unchanged on entry to Debug state. The unchanged registers include the IFSR, DFSR, DFAR, and IFAR.

On a processor that implements the Security Extensions, the SCR.NS bit is not changed on entry to Debug state.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

3. プロセッサは、デバッグ状態を開始したことを通知し、外部のエージェントが制御を獲得できるようになります。
   - DBGDSCR.HALTED ビットが 1 にセットされます。
   - P. C10-26 の表 C10-3 に従って DBGDSCR.MOE フィールドが設定されます。

推奨される外部デバッグインタフェースの詳細については、*実行制御およびクロストリガ信号* P. 付録 A-5 および *DBGACK および DBGCPUDONE* P. 付録 A-7 を参照して下さい。

C5.2.1 デバッグ状態への遷移が CP15 レジスタおよび DBGWFAR に及ぼす影響

デバッグ状態への遷移時に行われる動作は、デバッグ状態の開始された原因によって異なります。

- ウォッチポイントデバッグ イベントによってデバッグ状態に遷移した場合、ウォッチポイント対象のアドレスをアクセスした命令の命令仮想アドレス (IVA) に、デバッグイベントの生成時に設定されていたプロセッサの命令セット状態によって異なるオフセットを足した値で、DBGWFAR が更新されます。
  - ー ARM 状態では 8
  - ー Thumb および ThumbEE 状態では 4
  - ー Jazelle 状態では実装定義

  DBGWFAR の更新に使用される IVA の定義については、*メモリアドレス* P. C3-23 を参照して下さい。
- それ以外の場合、デバッグ状態への遷移時に DBGWFAR は変更されません。

注

- DBGWFAR の実装は、デバッグアーキテクチャのバージョンによって異なります。
  - ー v6 デバッグでは、CP15 c6 のレジスタとして実装されています。
  - ー v6.1 デバッグでは CP14 に実装されており、CP15 エイリアスの使用は推奨されません。
  - ー v7 デバックでは拡張 CP14 インタフェースに実装可能で、CP15 のエイリアスは存在しません。

  詳細については、*ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10-28 を参照して下さい。
- すべての場合について、デバッグ状態への遷移時に、DBGWFAR はこのセクションで説明されているように設定されます。

ARMv7 では、デバッグ状態への遷移時にすべての CP15 レジスタは変更されません。ARMv6 では、デバッグ状態への遷移時に DBGWFAR 以外の CP15 レジスタは変更されません。変更されないレジスタには、IFSR、DFSR、DFAR、IFAR が含まれます。

セキュリティ拡張機能を実装しているプロセッサでは、デバッグ状態への遷移時に SCR.NS ビットは変更されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.2.2 Asynchronous aborts and entry to Debug state

On entry to Debug state, it is IMPLEMENTATION DEFINED whether a processor ensures that all memory operations complete and that all possible outstanding asynchronous aborts have been recognized before it signals that it has entered Debug state.

Behavior in ARMv7

In ARMv7 the behavior on entry to Debug state is signaled by the value of the DBGDSCR.ADAdiscard bit:

If DBGDSCR.ADAdiscard == 1

The processor has already ensured that all possible outstanding asynchronous aborts have been recognized, and the debugger has no additional action to take.

If the processor logic always automatically sets DBGDSCR.ADAdiscard to 1 on entry to Debug state, then DBGDSCR.ADAdiscard is implemented as a read-only bit.

If DBGDSCR.ADAdiscard == 0

The following sequence must occur:

1. The debugger must execute an IMPLEMENTATION DEFINED sequence to determine whether all possible outstanding asynchronous aborts have been recognized.

   An asynchronous abort recognized as a result of this sequence is not acted on immediately. Instead, the processor latches the abort event and its type. The asynchronous abort is acted on when the processor leaves Debug state.

2. DBGDSCR.ADAdiscard is set to 1.

   There are two ways this requirement can be implemented:

   - The processor automatically sets this bit to 1 on detecting the execution of the IMPLEMENTATION DEFINED sequence. In this case, DBGDSCR.ADAdiscard is implemented as a read-only bit.
   - The IMPLEMENTATION DEFINED sequence sets DBGDSCR.ADAdiscard to 1, using the processor interface to the debug resources. In this case, DBGDSCR.ADAdiscard is implemented as a read/write bit.

When the processor has completed all Non-debug state memory operations it signals this to the system. It is IMPLEMENTATION DEFINED whether the processor ensures that all Non-debug state memory operations are complete on entry to Debug state. If not, the processor does not signal the system until all Non-debug state memory operations are complete. This might be linked to the debugger executing the IMPLEMENTATION DEFINED sequence to determine whether all possible outstanding asynchronous aborts have been recognized.

Details of the method used to signal to the system that Non-debug state memory operations are complete, including whether any such method is implemented, are IMPLEMENTATION DEFINED.

While the processor is in Debug state and DBGDSCR.ADAdiscard is 1, any memory access that causes an asynchronous abort has the effect of setting DBGDSCR.ADABORT_l, the Sticky Asynchronous Data Abort bit, to 1, but has no other effect on the state of the processor. The cause and type of the abort are not recorded. Because the abort is not pended, if the asynchronous abort is an external asynchronous abort and

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.2.2 非同期アボートとデバッグ状態への遷移

デバッグ状態への遷移時に、プロセッサがデバッグ状態に入ったことを通知する前に、すべてのメモリ操作が完了し、可能な未解決の非同期アボートがすべて認識されたことを保証するかどうかは実装定義です。

ARMv7 での動作

ARMv7 では、デバッグ状態への遷移時の動作は、DBGDSCR.ADAdiscard ビットの値によって通知されます。

DBGDSCR.ADAdiscard == 1 の場合

プロセッサにより、可能な未解決の非同期アボートがすべて認識されたことが保証されており、デバッガが追加の動作を行う必要はありません。

プロセッサのロジックで、デバッグ状態の開始時に DBGDSCR.ADAdiscard が常に自動的に 1 にセットされる場合、DBGDSCR.ADAdiscard は読み出し専用ビットとして実装されます。

DBGDSCR.ADAdiscard == 0 の場合

次のシーケンスが実行される必要があります。

1. デバッガは、可能な未解決の非同期アボートがすべて認識されたかどうかを判断するため、実装定義のシーケンスを実行する必要があります。

   このシーケンスの結果として認識された非同期アボートに対する動作は、直ちには行われません。代わりに、プロセッサはアボートイベントとそのタイプをラッチします。非同期アボートに対する動作は、プロセッサがデバッグ状態を終了したときに実行されます。

2. DBGDSCR.ADAdiscard が 1 にセットされます。

   この要件を実装する方法は 2 つあります。

   - プロセッサが実装定義のシーケンスの実行を検出した場合、このビットを自動的に 1 にセットします。この場合、DBGDSCR.ADAdiscard は読み出し専用ビットとして実装されます。
   - デバッグリソースへのプロセッサインタフェースを使用して、実装定義のシーケンスにより DBGDSCR.ADAdiscard を 1 にセットします。この場合、DBGDSCR.ADAdiscard は読み出し / 書き込みビットとして実装されます。

プロセッサは、非デバッグ状態のメモリ操作をすべて完了すると、そのことをシステムに通知します。デバッグ状態への遷移時に、非デバッグ状態のメモリ操作がすべて完了していることをプロセッサが保証するかどうかは実装定義です。保証しない場合、非デバッグ状態のメモリ操作がすべて完了するまで、プロセッサはシステムに対して通知を行いません。この場合、可能な未解決の非同期アボートがすべて認識されたかどうかを判別するため、デバッガが実装定義のシーケンスを実行する動作にリンクする可能性があります。

非デバッグ状態のメモリ操作が完了したことをシステムに通知するため使用される手法の詳細は、そのような手法を実装するかどうかも含めて、実装定義です。

プロセッサがデバッグ状態で、DBGDSCR.ADAdiscard が 1 の場合、非同期アボートを引き起こすすべてのメモリアクセスは、スティッキーな非同期データアボート ビットである DBGDSCR.ADABORT_l を 1 にセットする効果がありますが、プロセッサの状態に他の影響は及

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

the Interrupt Status Register (ISR) is implemented, the ISR.A bit is not updated. For more information, see *c12, Interrupt Status Register (ISR)* on page B3-150. The ISR is implemented only on processors that include the Security Extensions.

Any asynchronous abort that is latched before or during the entry to Debug state sequence is not overwritten by any new asynchronous abort. This means the latched abort is not discarded if the processor detects another asynchronous abort while DBGDSCR.ADAdiscard is set to 1. The processor acts on the latched abort on exit from Debug state. If the asynchronous abort is an external asynchronous abort and the ISR is implemented, the ISR.A bit reads as 1 indicating that an external abort is pending.

If the debugger has executed any memory access instructions, before exiting Debug state it must issue an IMPLEMENTATION DEFINED sequence of operations to ensure that any asynchronous aborts have been recognized and discarded.

On exit from Debug state, the processor automatically clears DBGDSCR.ADAdiscard to 0.

If an asynchronous abort is signalled to the processor before entry to Debug state or between entry to Debug state and DBGDSCR.ADAdiscard transitioning from 0 to 1, then the processor acts on the asynchronous abort on exit from Debug state:

- if the CPSR.A bit is 1, the abort is pended, and is taken when the A bit is cleared to 0
- if the CPSR.A bit is 0, the abort is taken by the processor.

For details of the recommended external debug interface, see *Run-control and cross-triggering signals* on page AppxA-5 and *DBGACK and DBGCPUDONE* on page AppxA-7.

Behavior in ARMv6

The behavior of asynchronous aborts on entry to Debug state differs between v6 Debug and v6.1 Debug:

v6 Debug DBGDSCR.ADAdiscard bit is not defined. A debugger must always perform a Data Synchronization Barrier (DSB) following entry to Debug state.

If the CPSR.A bit is 0 and an asynchronous abort is signalled, the processor takes a Data Abort exception as described in *Undefined Instruction and Data Abort exceptions in Debug state in v6 Debug* on page C5-23. A subsequent read of the processor state by the debugger returns the updated values of CPSR, LR_abt and SPSR_abt.

The value of DBGDSCR.ADABORT_l is UNKNOWN when in Non-debug state.

v6.1 Debug A debugger must always perform a DSB following entry to Debug state. This DSB causes DBGDSCR.ADAdiscard to be set to 1.

DBGDSCR.ADABORT_l is set to 1 on any asynchronous abort detected while the processor is in Debug state, regardless of the setting of DBGDSCR.ADAdiscard.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ぼしません。アボートの原因とタイプは記録されません。アボートは保留されないため、その非同期アボートが外部非同期アボートであり、割り込みステータスレジスタ (ISR) が実装されている場合、ISR.A ビットは更新されません。詳細については、*c12、割り込みステータスレジスタ (ISR)* P. B3-150 を参照して下さい。ISR は、セキュリティ拡張機能を含むプロセッサにのみ実装されています。

デバッグ状態シーケンスの開始前、または開始時にラッチされた非同期アボートは、新しい非同期アボートによって上書きされることはありません。つまり、ラッチされたアボートは、DBGDSCR.ADAdiscard が 1 にセットされている間にプロセッサが別の非同期アボートを検出した場合にも破棄されません。プロセッサは、デバッグ状態の終了時に、ラッチされたアボートに対応します。その非同期アボートが外部非同期アボートで、かつ ISR が実装されている場合、ISR.A ビットの読み出し値は 1 となり、外部アボートが保留中であることを示します。

デバッガがメモリアクセス命令を実行した場合、デバッグ状態を終了する前に実装定義の動作シーケンスを発行し、すべての非同期アボートが認識され、破棄されたことを保証する必要があります。

デバッグ状態の終了時に、プロセッサは DBGDSCR.ADAdiscard を自動的に 0 にクリアします。

デバッグ状態への遷移前、またはデバッグ状態への遷移から DBGDSCR.ADAdiscard が 0 から 1 に遷移するまでの間に、非同期アボートがプロセッサへ通知された場合、プロセッサはデバッグ状態の終了時に、その非同期アボートに対する動作を行います。

- CPSR.A ビットが 1 の場合、アボートは保留され、A ビットが 0 にクリアされた時点で取得されます。
- CPSR.A ビットが 0 の場合、アボートはプロセッサにより取得されます。

推奨される外部デバッグインタフェースの詳細については、*実行制御およびクロストリガ信号* P. 付録 A-5 および *DBGACK および DBGCPUDONE* P. 付録 A-7 を参照して下さい。

ARMv6 での動作

デバッグ状態開始時の非同期アボートの動作は、v6 デバッグと v6.1 デバッグとでは異なります。

v6 デバッグ DBGDSCR.ADAdiscard ビットは定義されていません。デバッガは、デバッグ状態への遷移後に必ずデータ同期バリア (DSB) を実行する必要があります。

CPSR.A ビットが 0 で、非同期アボートが通知された場合、プロセッサは *v6 デバッグのデバッグ状態での未定義命令およびデータアボート例外* P. C5-23 に説明されているようにデータアボート例外を取得します。デバッガによる以後のプロセッサ状態読み出しでは、CPSR、LR_abt、SPSR_abt の更新された値が返されます。

非デバッグ状態での DBGDSCR.ADABORT_l の値は未知です。

v6.1 デバッグ デバッガは、デバッグ状態への遷移後に必ず DSB を実行する必要があります。この DSB により、DBGDSCR.ADAdiscard は 1 にセットされます。

DBGDSCR.ADABORT_l は、DBGDSCR.ADAdiscard の設定にかかわらず、プロセッサがデバッグ状態の間に検出されたすべての非同期アボートについて、1 にセットされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.3 Behavior of the PC and CPSR in Debug state

Processing is halted on entry to Debug state, see *Entering Debug state* on page C5-3. After the processor has entered Debug state, a read of the PC returns a return address plus an offset. The return address depends on the type of debug event, and the offset depends on the instruction set state of the processor when Debug state was entered. Table C5-1 shows the values returned by a read of the PC.

Table C5-1 PC value while in Debug state

Debug event	PC value, for instruction set state on Debug entry			Meaning of return address (RA)[a] obtained from PC read
	ARM	Thumb or ThumbEE	Jazelle[b]	
Breakpoint	RA + 8	RA + 4	RA + Offset	Breakpointed instruction address
Synchronous Watchpoint	RA + 8	RA + 4	RA + Offset	Address of the instruction that triggered the watchpoint[c]
Asynchronous Watchpoint	RA + 8	RA + 4	RA + Offset	Address of the instruction for the execution to resume[d]
BKPT instruction	RA + 8	RA + 4	RA + Offset	`BKPT` instruction address
Vector Catch	RA + 8	RA + 4	RA + Offset	Vector address
External Debug Request	RA + 8	RA + 4	RA + Offset	Address of the instruction for the execution to resume
Halt Request	RA + 8	RA + 4	RA + Offset	Address of the instruction for the execution to resume
OS Unlock Catch	RA + 8	RA + 4	RA + Offset	Address of the instruction for the execution to resume

a. Return address (RA) is the address of the first instruction that the processor must execute on exit from Debug state. This enables program execution to continue from where it stopped.
b. *Offset* is an IMPLEMENTATION DEFINED value that is constant and documented.
c. Returning to RA has the effect of retrying the instruction. This can have implications under the memory order model. See *Synchronous and Asynchronous Watchpoint debug events* on page C3-18.
d. RA is not the address of the instruction that triggered the watchpoint, but one that was executed some number of instructions later. The address of the instruction that triggered the watchpoint can be discovered from the value in the DBGWFAR. See *Watchpoint Fault Address Register (DBGWFAR)* on page C10-28.

On entry to Debug state, the value of the CPSR is the value that the instruction at the return address would have been executed with, if it had not been cancelled by the debug event.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C5.3 デバッグ状態での PC および CPSR の動作

デバッグ状態への遷移時に処理は停止されます。*デバッグ状態への遷移* P. C5-3 を参照して下さい。プロセッサがデバッグ状態に遷移した後で PC を読み出すと、復帰アドレスにオフセットを足した値が返されます。復帰アドレスはデバッグイベントのタイプによって異なり、オフセットはデバッグ状態への遷移時にプロセッサが設定されていた命令セット状態によって異なります。PC の読み出しによって返される値を、表 C5-1 に示します。

表 C5-1 デバッグ状態での PC の値

デバッグイベント	デバッグ開始時の命令セット状態と PC の値との関係			PC 読み出しにより取得される復帰アドレス (RA)[a] の意味
	ARM	Thumb または ThumbEE	Jazelle[b]	
ブレークポイント	RA + 8	RA + 4	RA + オフセット	ブレークポイントが設定された命令のアドレス
同期ウォッチポイント	RA + 8	RA + 4	RA + オフセット	ウォッチポイントをトリガした命令のアドレス[c]
非同期ウォッチポイント	RA + 8	RA + 4	RA + オフセット	実行を再開する命令のアドレス[d]
BKPT 命令	RA + 8	RA + 4	RA + オフセット	`BKPT` 命令のアドレス
ベクタキャッチ	RA + 8	RA + 4	RA + オフセット	ベクタアドレス
外部デバッグ要求	RA + 8	RA + 4	RA + オフセット	実行を再開する命令のアドレス
ホールト要求	RA + 8	RA + 4	RA + オフセット	実行を再開する命令のアドレス
OS ロック解除キャッチ	RA + 8	RA + 4	RA + オフセット	実行を再開する命令のアドレス

a. 復帰アドレス (RA) は、デバッグ状態の終了時にプロセッサが実行しなければならない最初の命令のアドレスです。これによって、プログラムの実行が停止された場所から実行を再開することができます。
b. *オフセット*は実装定義の値で、一定であり文書化されます。
c. RA への復帰により、命令の再試行が行われます。これはメモリオーダ モデルに影響する場合があります。詳細については、*同期と非同期のウォッチポイントデバッグ イベント* P. C3-18 を参照して下さい。
d. RA は、ウォッチポイントをトリガした命令のアドレスではなく、そのいくつか後で実行された命令です。ウォッチポイントをトリガした命令のアドレスは、DBGWFAR の値から特定できます。詳細については、*ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10-28 を参照して下さい。

デバッグ状態への遷移時に、CPSR の値は、復帰アドレスの命令がデバッグ イベントによってキャンセルされなかったと仮定した場合に、その命令の実行に使用される値です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

This rule also applies to the CPSR.IT bits. On entry to Debug state these bits apply to the instruction at the return address.

The behavior of the PC and CPSR registers in Debug state is:

- The PC does not increment on instruction execution.
- The CPSR.IT status bits do not change on instruction execution.
- Predictable instructions that explicitly modify the PC or CPSR operate normally, updating the PC or CPSR.
- After the processor has entered Debug state, if 0b1111 (the PC) is specified as a source operand for an instruction it returns a value as described in Table C5-1 on page C5-7. The value read from the PC is aligned according to the rules of the instruction set state indicated by the CPSR.J and CPSR.T execution state bits, regardless of the fact that the processor only executes the ARM instruction set in Debug state. For more information, see *Executing instructions in Debug state* on page C5-9.
- If an instruction sequence for writing a particular value to the PC is executed while in Debug state, and the processor is later forced to restart without any additional write to the PC or CPSR, the execution starts at the address corresponding to the written value.
- If the CPSR is written to while in Debug state, subsequent reads of the PC return an UNKNOWN value, and if the processor is later forced to restart without having performed a write to the PC, the restart address is UNKNOWN. However, the CPSR can be read correctly while in Debug state.

  Note

  In v6 Debug, the CPSR and PC can be written in a single instruction, for example, `MOVS pc,lr`. In this case, the behavior is as if the CPSR is written first, followed by the PC. That is, if the processor is later forced to restart the restart address is predictable. This does not apply to v6.1 Debug or v7 Debug because in these versions of the Debug architecture such instructions are themselves UNPREDICTABLE in Debug state.

- If the processor is forced to restart without having performed a write to the PC, the restart address is UNKNOWN.
- If the PC is written to while in Debug state, later reads of the PC return an UNKNOWN value.

See also *Executing instructions in Debug state* on page C5-9, for more restrictions on instructions that might be executed in Debug state, including those that access the PC and CPSR.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

この規則は、CPSR.IT ビットにも適用されます。デバッグ状態への遷移時に、これらのビットは復帰アドレスの命令に適用されます。

デバッグ状態での PC および CPSR レジスタの動作は次のとおりです。

- PC は命令実行によりインクリメントされません。
- CPSR.IT ステータスビットは、命令実行により変更されません。
- PC または CPSR を明示的に変更する予測可能な命令は通常に動作し、PC または CPSR を更新します。
- プロセッサがデバッグ状態に遷移した後で、命令のソースオペランドとして 0b1111 (PC) が指定された場合、P. C5-7 の表 C5-1 に示されている値が返されます。PC から読み出される値は、CPSR.J および CPSR.T 実行状態ビットにより示される命令セット状態の規則に従ったアライメントです。デバッグ状態のプロセッサは ARM 命令セットのみを実行できるという事実は、この値には影響しません。詳細については、*デバッグ状態での命令の実行* P. C5-9 を参照して下さい。
- PC に特定の値を書き込む命令シーケンスがデバッグ状態で実行され、その後で PC または CPSR への追加書き込みなしにプロセッサが強制的に再始動された場合、書き込まれた値に対応するアドレスから実行が開始されます。
- デバッグ状態の間に CPSR への書き込みが行われ、その後で PC が読み出された場合、返される値は未知です。また、その後で PC への書き込みが実行されずに、プロセッサが強制的に再始動された場合、再始動アドレスは未知です。ただし、CPSR はデバッグ状態でも正しく読み出せます。

  注

  v6 デバッグでは、単一の命令、例えば `MOVS pc,lr` で CPSR と PC に書き込むことができます。この場合、CPSR に最初の書き込みが行われ、その後で PC が書き込まれたかのように動作します。つまり、その後でプロセッサが強制的に再始動された場合も、再始動アドレスは予測可能です。この動作は v6.1 デバッグまたは v7 デバッグには適用されません。これらのバージョンのデバッグアーキテクチャでは、このような命令自体がデバッグ状態で予測不能となるためです。

- PC への書き込みが行われず、プロセッサが強制的に再始動された場合、再始動アドレスは未知です。
- デバッグ状態で PC に書き込みが行われた場合、その後で PC を読み出したときに返される値は未知です。

PC および CPSR にアクセスする命令を含め、デバッグ状態で実行される可能性がある命令に関するその他の制限事項については、*デバッグ状態での命令の実行* P. C5-9 も参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.4 Executing instructions in Debug state

In Debug state the processor executes instructions issued through the Instruction Transfer Register, see *Instruction Transfer Register (DBGITR)* on page C10-46. This mechanism is enabled through DBGDSCR[13], see *Debug Status and Control Register (DBGDSCR)* on page C10-10.

The following rules and restrictions apply to instructions that can be executed in this manner in Debug state:

- The processor instruction set state always corresponds to the state indicated by the CPSR.J and CPSR.T execution state bits. However, the processor always interprets the instructions issued through the DBGITR as ARM instruction set opcodes, regardless of the setting of the CPSR.J and CPSR.T execution state bits.

  Some ARM instructions are UNPREDICTABLE if executed in Debug state. These instructions are either:

  — identified as UNPREDICTABLE in this list

  — shown as UNPREDICTABLE in Table C5-2 on page C5-10.

  Otherwise, except for the value read from the PC, instructions executed in Debug state operate as specified for ARM state. *Behavior of the PC and CPSR in Debug state* on page C5-7 specifies the value read from the PC.

- The CPSR.IT execution state bits are ignored. This means that instructions issued through the DBGITR do not fail their condition tests unexpectedly. However, the condition code field in an ARM instruction is honored.

  The CPSR.IT execution state bits are preserved and do not change when instructions are executed, unless an instruction that modifies those bits explicitly is executed.

- The branch instructions `B`, `BL`, `BLX` (immediate), and `BLX` (register) are UNPREDICTABLE in Debug state.

- The hint instructions `WFI`, `WFE` and `YIELD` are UNPREDICTABLE in Debug state.

- All memory read and memory write instructions with the PC as the base address register read an UNKNOWN value for the base address.

- Certain instructions that normally update the CPSR can be UNPREDICTABLE in Debug state, see *Writing to the CPSR in Debug state* on page C5-10.

- Instructions that load a value from memory into the PC are UNPREDICTABLE in Debug state.

- Conditional instructions that write explicitly to the PC are UNPREDICTABLE in Debug state.

- There are additional restrictions on data-processing instructions that write to the PC. See *Data-processing instructions with the PC as the target in Debug state* on page C5-12.

- The exception-generating instructions `SVC`, `SMC` and `BKPT` are UNPREDICTABLE in Debug state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C5.4 デバッグ状態での命令の実行

デバッグ状態では、プロセッサは命令転送レジスタ経由で発行された命令を実行します。*命令転送レジスタ (DBGITR)* P. C10-46 を参照して下さい。この機構は、DBGDSCR[13] により稼動されます。*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。

次の規則と制限事項は、デバッグ状態においてこの方法で実行可能な命令に適用されます。

- プロセッサの命令セット状態は、CPSR.J および CPSR.T 実行状態ビットにより示される状態に常に対応しています。ただし、プロセッサは CPSR.J および CPSR.T 実行状態ビットの設定にかかわらず、DBGITR 経由で発行される命令のすべてを ARM 命令セットのオペコードとして解釈します。

  一部の ARM 命令は、デバッグ状態で実行した場合は予測不能です。このような命令は次のいずれかのものです。

  – このリストで予測不能として示されている。

  – P. C5-10 の表 C5-2 で予測不能として示されている。

  それ以外の場合、PC から読み出される値を除いて、デバッグ状態で実行される命令は ARM 状態で指定されているように動作します。PC から読み出される値については、*デバッグ状態での PC および CPSR の動作* P. C5-7 を参照して下さい。

- CPSR.IT 実行状態ビットは無視されます。つまり、DBGITR 経由で発行された命令が、予期せずに条件テストに失敗することはありません。ただし、ARM 命令の条件コードフィールドは守られます。

  CPSR.IT 実行状態ビットは保持され、命令の実行時に変更されません。ただし、これらのビットを明示的に変更する命令が実行された場合を除きます。

- 分岐命令 `B`、`BL`、`BLX`（イミディエート）、`BLX`（レジスタ）は、デバッグ状態では予測不能です。

- ヒント命令 `WFI`、`WFE`、`YIELD` は、デバッグ状態では予測不能です。

- すべてのメモリ読み出しおよびメモリ書き込み命令において、PC をベースアドレス レジスタとして使用する場合、ベースアドレスとして読み出される値は未知です。

- 通常は CPSR を更新する特定の命令は、デバッグ状態では予測不能です。*デバッグ状態での CPSR への書き込み* P. C5-10 を参照して下さい。

- メモリから PC へ値をロードする命令は、デバッグ状態では予測不能です。

- PC への明示的な書き込みを行う条件付き命令は、デバッグ状態では予測不能です。

- PC への書き込みを行うデータ処理命令については、追加の制限事項があります。*PC をターゲットとするデータ処理命令のデバッグ状態での動作* P. C5-12 を参照して下さい。

- 例外を生成する命令 `SVC`、`SMC`、`BKPT` は、デバッグ状態では予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- A coprocessor can impose additional constraints or usage guidelines for executing coprocessor instructions in Debug state. For example a coprocessor that signals internal exception conditions asynchronously using the Undefined Instruction exception, as described in *Undefined Instruction exception* on page B1-49, might require particular sequences of instructions to avoid the corruption of coprocessor state associated with the exception condition.

  In the case of the VFP coprocessors, these sequences are defined by the VFP subarchitecture. Other coprocessors must define any sequences that they require.

Note

The definition of UNPREDICTABLE implies that an UNPREDICTABLE instruction executed in Debug state must not put the processor into a state or mode in which debug is not permitted, or change the state of any register that cannot be accessed from the current state and mode.

C5.4.1 Writing to the CPSR in Debug state

Table C5-2 lists all the instructions that normally update the CPSR, and shows their behavior in Debug state. Which instructions are permitted in Debug state depends on the version of the Debug architecture.

Table C5-2 Instructions that modify the CPSR, and their behavior in Debug state

Instruction	v6 Debug	v6.1 Debug, v7 Debug
BX	UNPREDICTABLE if CPSR.J is 1. Can be used to set or clear the CPSR.T bit.	UNPREDICTABLE.
BXJ	UNPREDICTABLE if either CPSR.J or CPSR.T is 1. Can be used to set CPSR.J to 1.	UNPREDICTABLE.
SETEND	UNPREDICTABLE.	UNPREDICTABLE.
CPS	UNPREDICTABLE.	UNPREDICTABLE.
<op>S PC,<Rn>,<Rm>[a]	Can be used to set the CPSR to any value by copying it from the SPSR of the current mode.	UNPREDICTABLE.
<op> PC,<Rn>,<Rm>[a]	Do not update the CPSR.	See *Data-processing instructions with the PC as the target in Debug state* on page C5-12.
MSR CPSR_fsxc	Use for setting the CPSR bits other than the execution state bits.	Use for setting the CPSR to any value.
MSR CPSR_<not fsxc>	Use for setting the CPSR bits other than the execution state bits.	UNPREDICTABLE.
LDM (exception return), RFE	UNPREDICTABLE.	UNPREDICTABLE.

a. <op> is one of ADC, ADD, AND, ASR, BIC, EOR, LSL, LSR, MOV, MVN, ORR, ROR, RRX, RSB, RSC, SBC, or SUB.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- コプロセッサ命令をデバッグ状態で実行する場合には、追加の制約事項や使用ガイドラインが適用されることがあります。例えば、コプロセッサが未定義命令例外を使用して、内部例外条件を非同期に通知する場合、*未定義命令例外* P. B1-49 に説明されているように、例外条件に関連するコプロセッサ状態が破壊されることを防止するために、命令の特定のシーケンスが要求されることがあります。

  VFP コプロセッサの場合、これらのシーケンスは VFP サブアーキテクチャにより定義されます。他のコプロセッサでは、必要なシーケンスを定義する必要があります。

注

予測不能の定義は、デバッグ状態で予測不能な命令が実行された場合、プロセッサをデバッグが許可されない状態またはモードに変更したり、現在の状態とモードからアクセスできないレジスタの状態を変更したりしてはならないことを意味しています。

C5.4.1 デバッグ状態での CPSR への書き込み

通常は CPSR を更新するすべての命令について、デバック状態での動作を表 C5-2 に示します。
デバッグ状態でどの命令が許可されるかは、デバッグアーキテクチャのバージョンにより異なります。

表 C5-2 CPSR を変更する命令と、そのデバッグ状態での動作

命令	v6 デバッグ	v6.1 デバッグ、v7 デバッグ
BX	CPSR.J が 1 の場合は予測不能。CPSR.T ビットをセットまたはクリアするために使用できます。	予測不能
BXJ	CPSR.J または CPSR.T が 1 の場合は予測不能。CPSR.J を 1 にセットするために使用できます。	予測不能
SETEND	予測不能	予測不能
CPS	予測不能	予測不能
<op>S PC,<Rn>,<Rm>[a]	現在のモードの SPSR から値をコピーすることで、CPSR を任意の値に設定できます。	予測不能
<op> PC,<Rn>,<Rm>[a]	CPSR を更新しません。	*PC をターゲットとするデータ処理命令のデバッグ状態での動作* P. C5-12 を参照して下さい。
MSR CPSR_fsxc	実行状態ビット以外の CPSR ビットを設定するために使用します。	CPSR を任意の値に設定するために使用します。
MSR CPSR_<not fsxc>	実行状態ビット以外の CPSR ビットを設定するために使用します。	予測不能
LDM(例外からの復帰)、RFE	予測不能	予測不能

a. <op> は、ADC、ADD、AND、ASR、BIC、EOR、LSL、LSR、MOV、MVN、ORR、ROR、RRX、RSB、RSC、SBC、SUB のいずれかです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

Table C5-2 on page C5-10 does not:

- Include instructions that only update the CPSR bits that are available in the APSR, that is the N, Z, C, V, Q, and GE[3:0] bits. These instructions have their normal behavior when executed in Debug state.
- Include instructions that cause exceptions, such as SVC, SMC, and memory access instructions that cause aborts. The behavior of these instructions is described in *Exceptions in Debug state* on page C5-20.
- Show what values can be written to the CPSR. For more information, see *Altering CPSR privileged bits in Debug state* on page C5-14.

MRS and MSR instructions in Debug state, in v6.1 Debug and v7 Debug

In v6.1 Debug and v7 Debug, if the debugger has to update bits in the CPSR that are not available in the APSR then it must use the MSR instruction to do so, writing to CPSR_fsxc. The behavior of the CPSR forms of the MSR and MRS instructions in Debug state differs from their behavior in Non-debug state. In the CPSR:

- in Non-debug state:
  - the execution state bits, other than the E bit, are RAZ when read by an MRS instruction
  - writes to the execution state bits, other than the E bit, by an MSR instruction are ignored
- in Debug state:
  - the execution state bits return their correct values when read by an MRS instruction
  - writes to the execution state bits by an MSR instruction update the execution state bits.

MRS and MSR instructions that read and write an SPSR behave as they do in Non-debug state.

In addition, in Debug state in v6.1 Debug and v7 Debug:

- if you use an MSR instruction to directly modify the execution state bits of the CPSR, you must then perform an Instruction Synchronization Barrier (ISB) operation
- an MSR instruction that does not write to all fields of the CPSR is UNPREDICTABLE
- if an MRS instruction reads the CPSR after an MSR writes the execution state bits, and before an ISB, the value returned is UNKNOWN
- if the processor leaves Debug state after an MSR writes the execution state bits, and before an ISB, the behavior of the processor is UNPREDICTABLE.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

P. C5-10 の表 C5-2 には、次の内容は含まれていません。

- APSR に存在する CPSR ビット、つまり N、Z、C、V、Q、および GE[3:0] ビットのみを更新する命令。これらの命令は、デバッグ状態で実行されたときも通常の動作を行います。
- SVC や SMC など例外を引き起こす命令と、アボートを引き起こすメモリアクセス命令。これらの命令の動作については、*デバッグ状態での例外* P. C5-20 を参照して下さい。
- CPSR に書き込むことができる値。詳細については、*デバッグ状態での CPSR 特権ビットの変更* P. C5-14 を参照して下さい。

v6.1 デバッグおよび v7 デバッグにおける、デバッグ状態での MRS および MSR 命令

v6.1 デバッグおよび v7 デバッグでは、APSR に存在しない CPSR ビットをデバッガで更新する必要がある場合、MSR 命令を使用して CPSR_fsxc に書き込む必要があります。MSR および MRS 命令の CPSR 形式の動作は、デバッグ状態では非デバッグ状態と異なります。CPSR については、次のような動作が行われます。

- 非デバッグ状態
  - E ビット以外の実行状態ビットは、MRS 命令によって RAZ として読み出されます。
  - E ビット以外の実行状態ビットへの MSR 命令による書き込みは無視されます。
- デバッグ状態
  - 実行状態ビットが MRS 命令によって読み出された場合、正しい値が返されます。
  - MSR 命令によって実行状態ビットへ書き込みが行われた場合、実行状態ビットが更新されます。

SPSR の読み出しおよび書き込みを行う MRS および MSR 命令は、非デバッグ状態と同じに動作します。

これらに加えて、v6.1 デバッグおよび v7 デバッグのデバッグ状態では、次の条件が適用されます。

- MSR 命令を使用して、CPSR の実行状態ビットを直接変更する場合、その後で命令同期バリア (ISB) 操作を実行する必要があります。
- CPSR のすべてのフィールドに書き込みを行わない MSR 命令は予測不能です。
- MSR 命令によって実行状態ビットを書き込んだ後で、ISB の前に MRS 命令で CPSR を読み出した場合、返される値は未知です。
- MSR 命令によって実行状態ビットを書き込んだ後で、ISB の前にプロセッサがデバッグ状態を終了した場合、プロセッサの動作は予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.4.2 Data-processing instructions with the PC as the target in Debug state

The ARM encodings of the instructions ADC, ADD, AND, ASR, BIC, EOR, LSL, LSR, MOV, MVN, ORR, ROR, RRX, RSB, RSC, SBC, and SUB write to the PC if their Rd field is 0b1111.

When in Non-debug state, these ARM instruction encodings can be executed only in the ARM instruction set state, and their behavior is described in:

- *SUBS PC, LR and related instructions* on page B6-25, if the S bit of the instruction is 1.
- Chapter A8 *Instruction Details*, if the S bit of the instruction is 0. These ARM instructions cause interworking branches in ARMv7, and simple branches in earlier versions of the architecture. The ALUWritePC() pseudocode function describes this operation, see *Pseudocode details of operations on ARM core registers* on page A2-12.

In Debug state, these ARM instruction encodings can be executed in any instruction set state, and the following additional restrictions apply:

- If the S bit of the instruction is 1:
  - — in v7 Debug and v6.1 Debug, behavior is UNPREDICTABLE
  - — in v6 Debug, behavior is as in Non-debug state.
- If the S bit of the instruction is 0, the behavior is always either a simple branch without changing instruction set state or UNPREDICTABLE. Table C5-3 shows how this behavior depends on the instruction set state, the value alu<1:0> written to the PC, and the architecture version.

Table C5-3 Debug state rules for data-processing instructions that write to the PC

CPSR.J	CPSR.T	Instruction set state	Architecture version	alu<1:0>	Operation[a]
0	0	ARM	ARMv7	00	BranchTo(alu<31:2>:'00')
				x1	UNPREDICTABLE[b]
				10	UNPREDICTABLE
			ARMv6	xx	BranchTo(alu<31:2>:'00')
X	1	Thumb or ThumbEE	ARMv7	x0	UNPREDICTABLE[b]
				x1	BranchTo(alu<31:1>:'0')
			ARMv6	xx	BranchTo(alu<31:1>:'0')
1	0	Jazelle	ARMv7 or ARMv6	xx	BranchTo(alu<31:0>)

a. Pseudocode description of behavior, when the behavior is not UNPREDICTABLE.
b. This behavior is changed from the behavior in Non-debug state. In all other rows, the behavior described is unchanged from the behavior in Non-debug state.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C5.4.2 PC をターゲットとするデータ処理命令のデバッグ状態での動作

ADC、ADD、AND、ASR、BIC、EOR、LSL、LSR、MOV、MVN、ORR、ROR、RRX、RSB、RSC、SBC、SUB の各命令の ARM エンコードでは、Rd フィールドが 0b1111 の場合に PC へ書き込みを行います。

非デバッグ状態では、これらの ARM 命令エンコードは ARM 命令セット状態でのみ実行可能です。命令の動作については次のセクションを参照して下さい。

- 命令の S ビットが 1 の場合、*SUBS PC*、*LR*、*および関連する命令* P. B6-25 を参照して下さい。
- 命令の S ビットが 0 の場合、第 A8 章*「命令の詳細」*を参照して下さい。これらの ARM 命令は、ARMv7 ではインターワーキング分岐を引き起こし、アーキテクチャの以前のバージョンでは単純な分岐を引き起こします。ALUWritePC() 擬似コード関数は、この動作を説明しています。*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 を参照して下さい。

デバッグ状態では、これらの ARM 命令エンコードはどの命令セット状態でも実行可能で、次のような追加の制限が適用されます。

- 命令の S ビットが 1 の場合
  - – v7 デバッグおよび v6.1 デバックでは、動作は予測不能です。
  - – v6 デバッグでは、非デバッグ状態と同じに動作します。
- 命令の S ビットが 0 の場合、命令セット状態を変更しない単純な分岐を行うか、予測不能な動作が引き起こされます。命令セット状態、PC に書き込まれる値の alu<1:0>、アーキテクチャのバージョンに対してどのような動作が行われるかを、表 C5-3 に示します。

表 C5-3 PC へ書き込みを行うデータ処理命令についてのデバッグ状態での規則

CPSR.J	CPSR.T	命令セット状態	アーキテクチャのバージョン	alu<1:0>	操作[a]
0	0	ARM	ARMv7	00	BranchTo(alu<31:2>:'00')
				x1	予測不能[b]
				10	予測不能
			ARMv6	xx	BranchTo(alu<31:2>:'00')
X	1	Thumb または ThumbEE	ARMv7	x0	予測不能[b]
				x1	BranchTo(alu<31:1>:'0')
			ARMv6	xx	BranchTo(alu<31:1>:'0')
1	0	Jazelle	ARMv7 または ARMv6	xx	BranchTo(alu<31:0>)

a. 動作が予測不能ではない場合、動作を擬似コードで説明しています。
b. この動作は、非デバッグ状態の動作とは異なります。他のすべての行では、動作は非デバッグ状態と同じです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.5 Privilege in Debug state

In Debug state, instructions issued to the processor have the privileges to access and modify processor registers, memory and coprocessor registers that they would have if issued in the same mode and security state in Non-debug state.

In User mode and Debug state, instructions have additional privileges to access or modify some registers and fields that cannot be accessed in User mode in Non-debug state. However, on processors that implement the Security Extensions and support Secure User halting debug, these additional privileges are restricted when all the following conditions are true:

- the processor is in Debug state
- the processor is in Secure User mode
- invasive debug is not permitted in Secure privileged modes, because either **DBGEN** or **SPIDEN** is LOW, see Chapter C2 *Invasive Debug Authentication*.

The following sections describe the instruction privileges, and the restrictions on them when these conditions are all true:

- *Accessing registers and memory in Debug state*
- *Altering CPSR privileged bits in Debug state* on page C5-14
- *Changing the SCR.NS bit in Debug state* on page C5-15
- *Coprocessor and Advanced SIMD instructions in Debug state* on page C5-16.

C5.5.1 Accessing registers and memory in Debug state

The rules for accessing ARM core registers and memory are the same in Debug state as in Non-debug state. For example, if the CPSR mode bits indicate the processor is in Supervisor mode:

- reads of ARM core registers return the Supervisor mode registers
- normal load and store operations make privileged accesses to memory
- a load or store with User mode privilege operation, for example `LDRT`, makes a User mode privilege access.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.5 デバッグ状態での特権

デバッグ状態では、プロセッサに対して発行される命令は、非デバッグ状態において同じモードとセキュリティ状態でその命令が発行された場合と同じ特権で、プロセッサレジスタ、メモリ、コプロセッサレジスタにアクセスし、変更できます。

ユーザモードのデバッグ状態では、非デバッグ状態のユーザモードではアクセスできないいくつかのレジスタとフィールドにアクセスまたは変更する追加特権が命令に与えられます。ただし、セキュリティ拡張機能を実装し、セキュアユーザ ホールトデバッグをサポートしているプロセッサでは、次の条件のすべてが真の場合には、これらの追加特権が制限されます。

- プロセッサがデバッグ状態である。
- プロセッサがセキュアユーザ モードである。
- DBGEN または SPIDEN が LOW であるため、セキュア特権モードで侵襲性デバッグが許可されていない。第 C2 章*「侵襲性デバッグの認証」*を参照して下さい。

次に示すセクションでは、命令の特権と、これらの条件がすべて真である場合の制限について説明しています。

- *デバッグ状態でのレジスタおよびメモリへのアクセス*
- *デバッグ状態での CPSR 特権ビットの変更* P. C5- 14
- *デバッグ状態での SCR.NS ビットの変更* P. C5- 15
- *デバッグ状態でのコプロセッサおよびアドバンスト SIMD 命令* P. C5- 16

C5.5.1 デバッグ状態でのレジスタおよびメモリへのアクセス

ARM コアレジスタおよびメモリにアクセスするための規則は、デバッグ状態でも非デバッグ状態と同じです。例えば、CPSR モードビットにより、プロセッサがスーパバイザモードであることが示されている場合、次の条件が適用されます。

- ARM コアレジスタを読み出すと、スーパバイザモードのレジスタが返されます。
- 通常のロードおよびストア操作では、メモリへの特権アクセスが行われます。
- ユーザモード特権でのロードまたはストア操作、例えば `LDRT` では、ユーザモードの特権アクセスが行われます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.5.2 Altering CPSR privileged bits in Debug state

On processors that implement the Security Extensions, the processor:

- prevents attempts to set the CPSR.M field to a value that would place the processor in a mode or security state where debug is not permitted
- prevents updates to the Privileged bits of the CPSR in cases where Secure User halting debug is supported, the processor is in Secure User mode, and invasive debug is not permitted in Secure privileged modes
- prevents attempts to set the CPSR.M field to 0b10001, FIQ mode, if NSACR.RFR == 1 and the processor is in Non-secure state.

On processors that do not implement the Security Extensions, all CPSR updates that are permitted in a privileged mode when in Non-debug state, are permitted in Debug state.

Table C5-4 defines the behavior on writes to the CPSR in Debug state.

Table C5-4 Permitted updates to the CPSR in Debug state

Mode	Secure state	Logical (DBGEN AND SPIDEN)	SU halting debug[a] supported	Update privileged CPSR bits[b]	Modify CPSR.M to Monitor mode
User	Yes	0	Yes	Update ignored	UNPREDICTABLE[c]
			No	Permitted[d]	Permitted
Privileged	Yes	0	X	Permitted[d]	Permitted
Any	No	0	X	Permitted[d]	UNPREDICTABLE[e]
Any	X	1	X	Permitted[d]	Permitted

a. Secure User halting debug support.
b. This column does not apply to changing CPSR.M to Monitor mode. Apart from this, the CPSR bits are defined in *Program Status Registers (PSRs)* on page B1-14, and this column does apply to changing CPSR.M to any other value.
c. The definition of UNPREDICTABLE implies the processor must not enter a privileged mode.
d. Except that, regardless of the state of **SPIDEN**:
The SCR.AW, SCR.FW and SCTLR.NMFI bits have the same effects on writes to CPSR.A and CPSR.F as they do in Non-debug state, see *Control of exception handling by the Security Extensions* on page B1-41 and *Non-maskable fast interrupts* on page B1-18.
The NSACR.RFR bit has the same effect on writes to CPSR.M as it does in Non-debug state, see *c1, Non-Secure Access Control Register (NSACR)* on page B3-110.
e. The definition of UNPREDICTABLE implies the processor must not enter Monitor mode, and must not enter FIQ mode when NSACR.RFR == 1.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C5.5.2 デバッグ状態での CPSR 特権ビットの変更

セキュリティ拡張機能を実装しているプロセッサでは、次の動作が行われます。

- CPSR.M フィールドに、デバッグが許可されないモードやセキュリティ状態にプロセッサを変更する値を設定する試みが禁止されます。
- セキュアユーザ ホールトデバッグがサポートされ、プロセッサがセキュアユーザ モードで、セキュア特権モードでの侵襲性デバッグが許可されていない場合に、CPSR の特権ビットの更新が禁止されます。
- NSACR.RFR == 1 かつプロセッサが非セキュア状態のとき、CPSR.M フィールドに 0b10001（FIQ モード）を設定する試みが禁止されます。

セキュリティ拡張機能を実装していないプロセッサでは、非デバッグ状態の特権モードで許可されるすべての CPSR 更新は、デバッグ状態でも許可されます。

デバッグ状態での CPSR への書き込み動作を、表 C5-4 に示します。

表 C5-4 デバッグ状態で許可される CPSR 更新

モード	セキュア状態	論理 (DBGEN AND SPIDEN)	SU ホールトデバッグ[a] のサポート	特権 CPSR ビットの更新[b]	CPSR.M のモニタモードへの変更
ユーザ	はい	0	はい	更新は無視されます。	予測不能[c]
			いいえ	許可[d]	許可
特権	はい	0	X	許可[d]	許可
任意	いいえ	0	X	許可[d]	予測不能[e]
任意	X	1	X	許可[d]	許可

a. セキュアユーザ ホールトデバッグのサポート
b. この列は、CPSR.M をモニタモードに変更する操作については適用されません。これ以外の CPSR ビットは*プログラムステータス レジスタ (PSR)* P. B1-14 に定義されており、CPSR.M を他の値に変更する操作にはこの列が適用されます。
c. 予測不能の定義では、プロセッサが特権モードに変更されてはならないことが条件です。
d. 例外として、SPIDEN の状態にかかわらず、次の条件が適用されます。
SCR.AW、SCR.FW、SCTLR.NMFI ビットは、CPSR.A および CPSR.F への書き込みに対して、非デバッグ状態と同じ影響を及ぼします。*セキュリティ拡張機能による例外処理の制御* P. B1-41 および*マスク不能高速割り込み* P. B1-18 を参照して下さい。
NSACR.RFR ビットは、CPSR.M への書き込みに対して、非デバッグ状態と同じ影響を及ぼします。*c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3-110 を参照して下さい。
e. 予測不能の定義では、プロセッサがモニタモードに入ってはならない、および NSACR.RFR == 1 の場合に FIQ モードに入ってはならないことが条件です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Being in Debug state when invasive halting debug is not permitted

A processor can be in a Secure privileged mode with **SPIDEN** LOW, see *Generation of debug events* on page C3-40 and *Changing the authentication signals* on page AppxA-4. More generally, it is possible to be in Debug state when the current mode, security state or debug authentication signals indicate that, in Non-debug state, debug events would be ignored. There are two situations where this can occur:

- Between a change in the debug authentication signals and the end of the next Instruction Synchronization Barrier operation, exception entry, or exception return. At this point it is it is UNPREDICTABLE whether the behavior of debug events that are generated follows the old or the new authentication signal settings.
- Because it is possible to change the authentication signals while in Debug state.

For example, the following sequence of events can occur:

1. The processor is in a Secure privileged mode. **SPIDEN** and **DBGEN** are both HIGH.
2. An instruction is prefetched that matches all the conditions for a breakpoint to occur.
3. That instruction is committed for execution.
4. At the same time, an external device writes to the peripheral that controls **SPIDEN** and **DBGEN**, causing **SPIDEN** to be deasserted to LOW.
5. **SPIDEN** changes, but the processor is already committed to entering Debug state.
6. The processor enters Debug state and is in a Secure privileged mode, even though **SPIDEN** is LOW.

If this series of events occurs, the processor can change to other Secure privileged modes, including Monitor mode, and update privileged bits in the CPSR, because it is in a privileged mode. However, if the processor leaves Secure state or moves to Secure User mode, it might not be able to return to a Secure privileged mode.

C5.5.3 Changing the SCR.NS bit in Debug state

SCR.NS is the Non-secure state bit, see *c1, Secure Configuration Register (SCR)* on page B3-106. Because this bit is part of a coprocessor register, the rules for executing coprocessor instructions in Debug state apply, see *Coprocessor and Advanced SIMD instructions in Debug state* on page C5-16.

In Debug state, the SCR can be written to:

- when Secure User halting debug is supported:
  - in any Secure privileged mode, including Monitor mode, regardless of the state of **DBGEN** and **SPIDEN**
  - in Secure User mode only if **DBGEN** and **SPIDEN** are both HIGH
- when Secure User halting debug is not supported, in any Secure mode, including Monitor mode, regardless of the state of **DBGEN** and **SPIDEN**.

A write to the SCR in any other case is treated as an Undefined Instruction exception. For details of how Undefined Instruction exceptions are handled in Debug state see *Exceptions in Debug state* on page C5-20.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

侵襲性ホールトデバッグが許可されていない場合のデバッグ状態

プロセッサは、SPIDEN が LOW のときにセキュア特権モードとなることがあります。*デバッグイベントの生成* P. C3-40 および*認証信号の変更* P. 付録 A-4 を参照して下さい。より一般的には、現在のモード、セキュリティ状態、デバッグ認証信号により、非デバッグ状態でデバッグイベントが無視されることが示されているときに、デバッグ状態になることがあります。この条件が発生する状況は 2 つあります。

- デバッグ認証信号の変化から、次の命令同期バリア操作、例外開始、または例外からの復帰が終了するまでの間。この期間は、生成されるデバッグイベントが、認証信号の古い設定と新しい設定のどちらに従って動作するかは予測不能です。
- プロセッサがデバッグ状態の間に認証信号を変化させることが可能なため。

例えば、次のようなイベントのシーケンスが発生する可能性があります。

1. プロセッサはセキュア特権モードです。SPIDEN と DBGEN はどちらも HIGH です。
2. ブレークポイントの発生条件のすべてに一致する命令がプリフェッチされます。
3. この命令が実行にコミットされます。
4. 同時に、外部デバイスが、SPIDEN と DBGEN を制御するペリフェラルに書き込みを行い、SPIDEN がアサート解除されて LOW になります。
5. SPIDEN が変更されますが、プロセッサはすでにデバッグ状態への遷移にコミットされています。
6. プロセッサは、SPIDEN が LOW であるにもかかわらず、デバッグ状態に遷移し、セキュア特権モードとなります。

この一連のイベントが発生した場合、プロセッサは特権モードであるため、モニタモードを含む他のセキュア特権モードに変更でき、CPSR の特権ビットを更新することもできます。しかし、プロセッサがセキュア状態を終了したり、セキュアユーザ モードに移行したりした場合、セキュア特権モードに戻れない可能性があります。

C5.5.3 デバッグ状態での SCR.NS ビットの変更

SCR.NS は非セキュア状態ビットです。*c1、セキュア構成レジスタ (SCR)* P. B3-106 を参照して下さい。このビットはコプロセッサレジスタの一部なので、デバッグ状態でコプロセッサ命令を実行するための規則が適用されます。*デバッグ状態でのコプロセッサおよびアドバンスト SIMD 命令* P. C5-16 を参照して下さい。

デバッグ状態では、次の条件で SCR への書き込みが可能です。

- セキュアユーザ ホールトデバッグがサポートされている場合
  - モニタモードを含むすべてのセキュア特権モードで、DBGEN と SPIDEN の状態にかかわらず。
  - セキュアユーザ モードで、DBGEN と SPIDEN の両方が HIGH の場合のみ。
- セキュアユーザ ホールトデバッグがサポートされていない場合、モニタモードを含む任意のセキュアモードで、DBGEN と SPIDEN の状態にかかわらず。

他のすべての場合、SCR への書き込みは未定義命令例外として扱われます。デバッグ状態で未定義命令例外がどのように扱われるかの詳細については、*デバッグ状態での例外* P. C5-20 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

This is a particular case of the rules for accessing CP15 registers described in *Coprocessor and Advanced SIMD instructions in Debug state*.

Note

Normally, in Monitor mode, any exception automatically clears the SCR.NS bit to 0. However an exception while in Debug state in Monitor mode does not have any effect on the value of the SCR.NS bit.

C5.5.4 Coprocessor and Advanced SIMD instructions in Debug state

The following sections describe the coprocessor and Advanced SIMD instructions in Debug state:

- *Instructions for CP0 to CP13, and Advanced SIMD instructions*
- *Instructions for CP14 and CP15* on page C5-17.

Instructions for CP0 to CP13, and Advanced SIMD instructions

This subsection describes:

- Coprocessor instructions for CP0 to CP13. These include the VFP instructions.
- If the Advanced SIMD extension is implemented, the instruction encodings described in *Advanced SIMD data-processing instructions* on page A7-10 and *Advanced SIMD element or structure load/store instructions* on page A7-27.

Access controls for these instructions are determined:

- by the CPACR, see:
  - *c1, Coprocessor Access Control Register (CPACR)* on page B3-104, for a VMSA implementation
  - *c1, Coprocessor Access Control Register (CPACR)* on page B4-51, for a PMSA implementation.
- additionally, if the Security Extensions are implemented, by the NSACR, see *c1, Non-Secure Access Control Register (NSACR)* on page B3-110.

In v6.1 Debug and v7 Debug, in Debug state the current mode and security state define the privilege and access controls for these instructions.

In v6 Debug, in Debug state it is IMPLEMENTATION DEFINED whether these instructions are executed using the privilege and access controls for the current mode and security state, or using the privilege and access controls for a privileged mode in the current security state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

これは、*デバッグ状態でのコプロセッサおよびアドバンスト SIMD 命令*に説明されている CP15 レジスタへアクセスする特別な場合です。

注

通常は、モニタモードでは、例外が発生すると SCR.NS ビットが自動的に 0 にクリアされます。しかし、モニタモードのデバック状態で例外が発生した場合、SCR.NS ビットの値は影響を受けません。

C5.5.4 デバッグ状態でのコプロセッサおよびアドバンスト SIMD 命令

次に示すセクションでは、デバッグ状態でのコプロセッサおよびアドバンスト SIMD 命令について説明しています。

- *CP0 ～ CP13 の命令とアドバンスト SIMD 命令*
- *CP14 および CP15 の命令* P. C5- 17

CP0 ～ CP13 の命令とアドバンスト SIMD 命令

このサブセクションでは、次のトピックについて説明します。

- CP0 ～ CP13 のコプロセッサ命令。VFP 命令も含まれます。
- アドバンスト SIMD 拡張機能が実装されている場合、命令のエンコードは*アドバンスト SIMD のデータ処理命令* P. A7- 10 および*アドバンスト SIMD のエレメントまたは構造ロード／ストア命令* P. A7- 27 に説明されているとおりです。

これらの命令について、アクセス制御は次のように決定されます。

- CPACR によって決定されます。次のセクションを参照して下さい。
  - *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B3- 104（VMSA 実装の場合）
  - *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B4- 51（PMSA 実装の場合）
- さらに、セキュリティ拡張機能が実装されている場合は、NSACR によっても決定されます。*c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3- 110 を参照して下さい。

v6.1 デバッグと v7 デバッグでは、デバッグ状態でのこれらの命令に関する特権とアクセス制御は、現在のモードとセキュリティ状態によって定義されます。

v6 デバッグでは、これらの命令がデバッグ状態で、現在のモードとセキュリティ状態に対応する特権とアクセス制御を使用して実行されるのか、それとも現在のセキュリティ状態の特権モードに対応する特権とアクセス制御を使用して実行されるのかは、実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Instructions for CP14 and CP15

This subsection describes the coprocessor instructions for the internal coprocessors CP14 and CP15.

The two groups of registers provided by CP14 are:

- The CP14 debug registers, accessed by MCR and MRC instructions with <opc1> == 0b000. Some of these registers can also be accessed by CP14 LDC and STC instructions.
- The CP14 non-debug registers, accessed by MCR and MRC instructions with <opc1> != 0b000. These include the trace registers.

Accesses to CP14 and CP15 are as follows:

- Instructions that access CP14 or CP15 registers that are permitted (not UNDEFINED) in User mode when in Non-debug state, are always permitted in Debug state.
- Instructions that access CP14 debug registers that are permitted (not UNDEFINED) in privileged modes when in Non-debug state are permitted in Debug state, regardless of the debug authentication and the processor mode and security state.
- If Secure User halting debug is supported, ARM recommends that certain CP15 instructions that a debugger requires to maintain memory coherency are permitted in Debug state regardless of debug permissions and the processor mode, see *Access to specific cache management functions in Debug state* on page C5-25.
- If the processor is in a privileged mode or the debugger can write to the CPSR.M bits to change to a privileged mode, then instructions that access CP14 or CP15 registers that are permitted (not UNDEFINED) in privileged modes when in Non-debug state are permitted in Debug state. If the processor is in User mode there is no requirement to change to a privileged mode first.

  Note

  — Two particular cases are where Security Extensions are not implemented and where Secure User halting debug is not supported. In these cases the CPSR.M bits can always be changed to a privileged mode and, therefore, the debugger is able to access all CP14 and CP15 registers at all times.

  — Except for accesses to the Baseline CP14 debug registers, ARM deprecates accessing any CP14 or CP15 register from User mode in Debug state if that register cannot be accessed from User mode in Non-debug state.

- In every case, permissions to access CP14 and CP15 registers while in Debug state are never greater than the permissions granted to any privileged mode when in Non-debug state in the current security state.
- If the processor is in Secure User mode and the debugger cannot write to the CPSR.M bits to change to a privileged mode, then any instruction that accesses a CP14 non-debug register or a CP15 register is not permitted (UNDEFINED) in Debug state if it is not permitted in Secure User mode in Non-debug state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP14 および CP15 の命令

このサブセクションでは、内部コプロセッサ CP14 および CP15 のコプロセッサ命令について説明します。

CP14 で提供されるレジスタは、次の 2 つのグループに分けられます。

- CP14 デバッグレジスタ。MCR および MRC 命令で、<opc1> = 0b000 によりアクセスされます。これらのレジスタの一部は、CP14 の LDC および STC 命令でもアクセス可能です。
- CP14 非デバッグレジスタ。MCR および MRC 命令で、<opc1> != 0b000 によりアクセスされます。これにはトレースレジスタも含まれます。

CP14 および CP15 へのアクセスには、次の条件が適用されます。

- 非デバッグ状態のユーザモードで許可されている（未定義ではない）、CP14 または CP15 レジスタにアクセスする命令は、デバッグ状態でも常に許可されます。
- 非デバッグ状態の特権モードで許可されている（未定義ではない）、CP14 デバッグレジスタにアクセスする命令は、デバッグ認証およびプロセッサモードとセキュリティ状態にかかわらず、デバッグ状態で許可されます。
- セキュアユーザ ホールトデバッグがサポートされている場合、メモリコヒーレンシを維持するためにデバッガが必要とする特定の CP15 命令を、デバッグアクセス許可とプロセッサモードにかかわらず、デバッグ状態で許可することをお勧めします。*デバッグ状態での特定のキャッシュ管理機能へのアクセス* P. C5- 25 を参照して下さい。
- プロセッサが特権モードであるか、デバッガが CPSR.M ビットへの書き込みにより特権モードへの変更を行える場合、非デバッグ状態の特権モードで許可されている（未定義ではない）、CP14 または CP15 レジスタにアクセスする命令は、デバッグ状態で許可されます。プロセッサがユーザモードの場合、先に特権モードへ変更する必要はありません。

  注

  – セキュリティ拡張機能が実装されていない場合と、セキュアユーザ ホールトデバッグがサポートされていない場合の 2 つは、特別なケースです。これらの場合、CPSR.M ビットは常に特権モードへの変更が可能なため、デバッガは常にすべての CP14 および CP15 レジスタにアクセスできます。

  – ベースライン CP14 デバッグレジスタの場合を除き、非デバッグ状態のユーザモードから CP14 または CP15 レジスタにアクセスできない場合、デバッグ状態のユーザモードからこれらのレジスタにアクセスすることは推奨されません。

- どのような場合も、デバッグ状態の間に CP14 および CP15 レジスタにアクセスするためのアクセス許可は、現在のセキュリティ状態で非デバッグ状態の間に、いずれかの特権モードに与えられるアクセス許可よりも大きくなることはありません。
- プロセッサがセキュアユーザ モードで、デバッガが CPSR.M ビットへの書き込みにより特権モードに変更できない場合、CP14 非デバッグレジスタまたは CP15 レジスタにアクセスする命令は、非デバッグ状態のセキュアユーザ モードで許可されていなければ、デバッグ状態でも許可されません（未定義）。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- Any CP14 or CP15 register access that is not permitted generates an Undefined Instruction exception. For details of how Undefined Instruction exceptions are handled in Debug state see *Exceptions in Debug state* on page C5-20.
- If the processor is in a privileged mode or the debugger can write to the CPSR.M bits to change to a privileged mode, then any CP14 or CP15 instruction is UNPREDICTABLE in Debug state if that instruction is UNPREDICTABLE in Non-debug state.
- On processors that implement the Security Extensions, any access to a Banked CP15 register accesses the copy for the current security state. If the processor is in Monitor mode, the Non-debug state rules for accessing CP15 registers in Monitor mode apply.

This means that, for example:

- If the processor is stopped in Non-secure state and invasive debug is not permitted in Secure privileged modes then the debugger has access only to those CP15 registers accessible in Non-secure state in Non-debug mode.
- If the processor is stopped with invasive debug permitted in Secure privileged modes then the debugger has access to all CP15 registers. If the processor is in Non-secure state, the debugger can switch the processor to Monitor mode to access the SCR.NS bit, to give access to all CP15 registers.

Invasive debug is permitted in Secure privileged modes when both **SPIDEN** and **DBGEN** are HIGH.

In Debug state. the **CP15SDISABLE** input to the processor operates in exactly the same way as in Non-debug state, see *The CP15SDISABLE input* on page B3-76:

- if **CP15SDISABLE** is HIGH, any operation affected by **CP15SDISABLE** in Non-debug state results in an Undefined Instruction exception in Debug state
- if **CP15SDISABLE** is LOW, it has no effect on any register access.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- 許可されていない CP14 または CP15 レジスタへのアクセスは、すべて未定義命令例外を生成します。デバッグ状態で未定義命令例外がどのように扱われるかの詳細については、*デバッグ状態での例外* P. C5-20 を参照して下さい。
- プロセッサが特権モードであるか、デバッガが CPSR.M ビットへの書き込みにより特権モードへの変更を行える場合、CP14 または CP15 命令が非デバッグ状態で予測不能であれば、デバッグ状態でも予測不能です。
- セキュリティ拡張機能を実装しているプロセッサでは、バンク CP15 レジスタへのアクセスはすべて、現在のセキュリティ状態におけるコピーにアクセスします。プロセッサがモニタモードの場合、モニタモードで CP15 レジスタにアクセスするための非デバッグ状態での規則が適用されます。

例としては、次のようになります。

- プロセッサが非セキュア状態で停止しており、セキュア特権モードで侵襲性デバッグが許可されていない場合、デバッガは非デバッグモードの非セキュア状態でアクセス可能な CP15 レジスタにのみアクセスできます。
- プロセッサが停止しており、セキュア特権モードで侵襲性デバッグが許可されている場合、デバッガはすべての CP15 レジスタにアクセスできます。プロセッサが非セキュア状態の場合、デバッガはプロセッサをモニタモードに切り替えて SCR.NS ビットにアクセスし、すべての CP15 レジスタへのアクセス許可を得ることができます。

SPIDEN と DBGEN の両方が HIGH の場合、セキュア特権モードで侵襲性デバッグが許可されています。

デバッグ状態で、プロセッサへの CP15SDISABLE 入力は、非デバッグ状態と全く同一に動作します。*CP15SDISABLE 入力* P. B3-76 を参照して下さい。

- CP15SDISABLE が HIGH の場合、非デバッグ状態で CP15SDISABLE の影響を受ける操作はすべて、デバッグ状態では未定義命令例外を引き起こします。
- CP15SDISABLE が LOW の場合、レジスタアクセスへの影響はありません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.6 Behavior of non-invasive debug in Debug state

If any non-invasive debug features exist, their behavior in Debug state is broadly the same as when non-invasive debug is not permitted. For details see *About non-invasive debug authentication* on page C7-2.

Note

When the DBGDSCR.DBGack bit, Force Debug Acknowledge, is set to 1 and the processor is in Non-debug state, the behavior of non-invasive debug features is IMPLEMENTATION DEFINED. However, in this case non-invasive debug features must behave either as if in Debug state or as if Non-debug state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C5.6 デバッグ状態での非侵襲性デバッグの動作

非侵襲性デバッグ機能が存在する場合、デバッグ状態での動作は、非侵襲性デバッグが許可されていない場合とほぼ同じです。詳細については、*非侵襲性デバッグの認証について*P. C7-2を参照して下さい。

注

DBGDSCR.DBGack（強制デバッグ応答）ビットが 1 にセットされており、プロセッサが非デバッグ状態の場合、非侵襲性デバッグ機能の動作は実装定義です。ただし、この場合に非侵襲性デバッグ機能は、デバッグ状態と同様に、または非デバッグ状態と同様に動作する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.7 Exceptions in Debug state

This section describes how exceptions are handled when the processor is in Debug state:

- exception handling is the same in Debug state in v7 Debug and v6.1 Debug, except for some slight differences in when asynchronous aborts are recognized
- there are some differences in exception handling in Debug state in v6 Debug, and these are indicated.

Exceptions are handled as follows when the processor is in Debug state:

Reset On a Reset exception, the processor leaves Debug state. The reset handler runs in Non-debug state, see *Reset* on page B1-48.

Note

This only applies to a reset that in Non-debug state would cause a Reset exception. It does not apply to a debug logic reset. For more information on debug logic reset, see *Recommended reset scheme for v7 Debug* on page C6-16.

Prefetch Abort

A Prefetch Abort exception cannot be generated because no instructions are prefetched in Debug state.

SVC The `SVC` instruction is UNPREDICTABLE.

SMC The `SMC` instruction is UNPREDICTABLE.

BKPT The `BKPT` instruction is UNPREDICTABLE.

Debug events Debug events are ignored in Debug state.

Interrupts IRQ and FIQ exceptions are disabled and not taken in Debug state.

Note

This behavior does not depend on the values of the I and F bits in the CPSR, and the value of these bits are not changed on entering Debug state.

However, if the *Interrupt Status Register* (ISR) is implemented, the ISR.I and ISR.F bits continue to reflect the values of the IRQ and FIQ inputs to the processor. For more information, see *c12, Interrupt Status Register (ISR)* on page B3-150.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C5.7 デバッグ状態での例外

このセクションでは、プロセッサがデバッグ状態のときに例外がどのように処理されるかについて説明します。

- v7 デバッグおよび v6.1 デバッグでは、例外処理はデバッグ状態でも同じですが、非同期アボートが認識された場合にのみいくつかの小さな相違点があります。
- v6 デバッグのデバッグ状態での例外処理にはいくつかの相違点があり、これらの相違点は示されています。

プロセッサがデバッグ状態のとき、例外は次のように処理されます。

リセット リセット例外が発生すると、プロセッサはデバッグ状態を終了します。リセットハンドラは非デバッグ状態で実行されます。*リセット* P. B1-48 を参照して下さい。

注

これは、非デバッグ状態でリセット例外を引き起こすリセットにのみ適用されます。デバッグロジックのリセットには適用されません。デバッグロジックのリセットの詳細については、*v7 デバッグで推奨されるリセット方式* P. C6-16 を参照して下さい。

プリフェッチアボート

デバッグ状態では命令のプリフェッチが行われないため、プリフェッチアボート例外が生成されることはありません。

SVC `SVC` 命令は予測不能です。

SMC `SMC` 命令は予測不能です。

BKPT `BKPT` 命令は予測不能です。

デバッグイベント デバックイベントは、デバッグ状態では無視されます。

割り込み IRQ および FIQ 例外はデバッグ状態では不可能で、取得されません。

注

この動作は CPSR の I および F ビットの値の影響を受けません。デバッグ状態の開始時に、これらのビットの値は変更されません。

ただし、*割り込みステータスレジスタ* (ISR) が実装されている場合、ISR.I および ISR.F ビットは、プロセッサの IRQ および FIQ 入力の値を反映し続けます。詳細については、*c12*、*割り込みステータスレジスタ (ISR)* P. B3-150 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Undefined Instruction

Undefined Instruction exceptions are generated for the same reasons in Debug state as in Non-debug state.

The behavior depends on the Debug architecture version:

v6.1 Debug, v7 Debug

When an Undefined Instruction exception is generated in Debug state, the processor takes the exception as follows:

- PC, CPSR, SPSR_und, LR_und, SCR.NS, and DBGDSCR.MOE are unchanged.
- The processor remains in Debug state.
- DBGDSCR.UND_l, the Sticky Undefined Instruction bit, is set to 1.

For more information, see the description of the UND_l bit in *Debug Status and Control Register (DBGDSCR)* on page C10-10.

v6 Debug See *Undefined Instruction and Data Abort exceptions in Debug state in v6 Debug* on page C5-23.

Synchronous data abort

Data Abort exceptions are generated by synchronous data aborts in Debug state. The behavior depends on the Debug architecture version:

v6.1 Debug, v7 Debug

When a Data Abort exception is generated by a synchronous data abort in Debug state, the processor takes the exception as follows:

- PC, CPSR, SPSR_abt, LR_abt, SCR.NS, and DBGDSCR.MOE are unchanged.
- The processor remains in Debug state.
- DBGDSCR.SDABORT_l, the Sticky Synchronous Data Abort bit, is set to 1.
- The DFSR and DFAR are updated if any of:
  - — Secure User halting debug is not supported
  - — the processor is not in Secure User mode
  - — invasive debug is permitted in Secure privileged modes.

  Otherwise it is IMPLEMENTATION DEFINED whether the DFSR and DFAR are updated.
- If the ISR is implemented, the ISR.A bit is not changed, because no abort is pended.

See also the description of the SDABORT_l bit in *Debug Status and Control Register (DBGDSCR)* on page C10-10.

v6 Debug See *Undefined Instruction and Data Abort exceptions in Debug state in v6 Debug* on page C5-23.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

未定義命令

未定義命令例外は、デバッグ状態でも非デバッグ状態と同じ原因で生成されます。

動作はデバッグアーキテクチャのバージョンによって異なります。

v6.1 デバッグ、v7 デバッグ

デバッグ状態で未定義命令例外が生成されると、プロセッサは次のように例外を取得します。

- PC、CPSR、SPSR_und、LR_und、SCR.NS、DBGDSCR.MOE は変更されません。
- プロセッサはデバッグ状態のままです。
- DBGDSCR.UND_l（スティッキー未定義命令）ビットは 1 にセットされます。

詳細については、*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 の UND_l ビットについての説明を参照して下さい。

v6デバッグ *v6 デバッグのデバッグ状態での未定義命令およびデータアボート例外* P. C5-23 を参照して下さい。

同期データアボート

データアボート例外は、デバッグ状態での同期データアボートにより生成されます。動作はデバッグアーキテクチャのバージョンによって異なります。

v6.1 デバッグ、v7 デバッグ

デバッグ状態での同期データアボートによりデータアボート例外が生成されると、プロセッサは次のように例外を取得します。

- PC、CPSR、SPSR_abt、LR_abt、SCR.NS、DBGDSCR.MOE は変更されません。
- プロセッサはデバッグ状態のままです。
- DBGDSCR.SDABORT_l（スティッキー同期データアボート）ビットは 1 にセットされます。
- 次のいずれかの条件が当てはまる場合、DFSR および DFAR が更新されます。
  - – セキュアユーザ ホールトデバッグがサポートされていない。
  - – プロセッサがセキュアユーザ モードではない。
  - – セキュア特権モードで侵襲性デバッグが許可されている。

  それ以外の場合、DFSR および DFAR が更新されるかどうかは実装定義です。
- ISR が実装されている場合、アボートが保留されないため、ISR.A ビットは変更されません。

デバッグステータスおよび制御レジスタ (DBGDSCR) P. C10-10 の SDABORT_l ビットについての説明も参照して下さい。

v6 デバッグ

v6 デバッグのデバッグ状態での未定義命令およびデータアボート例外 P. C5-23 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Asynchronous abort

The behavior depends on the Debug architecture version:

v6.1 Debug, v7 Debug

When an asynchronous abort is signalled in Debug state, no Data Abort exception is generated and the processor behaves as follows:

- The setting of the CPSR.A bit is ignored.
- PC, CPSR, SPSR_abt, LR_abt, SCR.NS, and DBGDSCR.MOE are unchanged.
- The processor remains in Debug state.
- The DFSR is unchanged.
- If DBGDSCR.ADAdiscard is 1:
  - DBGDSCR.ADABORT_l, the Sticky Asynchronous Data Abort bit, is set to 1.
  - On exit from Debug state, this asynchronous abort is not acted on.
  - If the ISR is implemented, the ISR.A bit is not changed, because no abort is pended.
- If DBGDSCR.ADAdiscard is 0:
  - In v7 Debug, DBGDSCR.ADABORT_l is unchanged.
  - In v6.1 Debug, DBGDSCR.ADABORT_l is set to 1.
  - On exit from Debug state, this asynchronous abort is acted on.
  - If the asynchronous abort is an external asynchronous abort, and the ISR is implemented, the ISR.A bit is set to 1 indicating that an external abort is pending.

See also:

- *Asynchronous aborts and entry to Debug state* on page C5-5.
- the descriptions of the ADABORT_l and ADAdiscard bits in *Debug Status and Control Register (DBGDSCR)* on page C10-10.

v6 Debug When an asynchronous abort is signalled in Debug state, then:

- if the CPSR.A bit is 0, the abort is generated when the CPSR.A bit is cleared to 0
- if the CPSR.A bit is 1, a Data Abort exception is generated, see *Undefined Instruction and Data Abort exceptions in Debug state in v6 Debug* on page C5-23.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

非同期アボート

動作はデバッグアーキテクチャのバージョンによって異なります。

v6.1 デバッグ、v7 デバッグ

デバッグ状態で非同期アボートが通知されると、データアボート例外は生成されず、プロセッサは次のように動作します。

- CPSR.A ビットの設定は無視されます。
- PC、CPSR、SPSR_abt、LR_abt、SCR.NS、DBGDSCR.MOE は変更されません。
- プロセッサはデバッグ状態のままです。
- DFSR は変更されません。
- DBGDSCR.ADAdiscard が 1 の場合
  - DBGDSCR.ADABORT_l（スティッキー非同期データアボート）ビットは 1 にセットされます。
  - デバッグ状態の終了時に、この非同期アボートに対する動作は実行されません。
  - ISR が実装されている場合、アボートが保留されないため、ISR.A ビットは変更されません。
- DBGDSCR.ADAdiscard が 0 の場合
  - v7 デバッグでは、DBGDSCR.ADABORT_l は変更されません。
  - v6.1 デバッグでは、DBGDSCR.ADABORT_l が 1 にセットされます。
  - デバッグ状態の終了時に、この非同期アボートに対する動作が実行されます。
  - その非同期アボートが外部非同期アボートで、かつ ISR が実装されている場合、ISR.A ビットは 1 にセットされ、外部アボートが保留中であることを示します。

次に示すセクションも参照して下さい。

- *非同期アボートとデバッグ状態への遷移* P. C5-5
- *デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 の、ADABORT_l および ADAdiscard ビットについての説明

v6 デバッグ

デバッグ状態で非同期アボートが通知されると、次の動作が行われます。

- CPSR.A ビットが 0 の場合、CPSR.A ビットが 0 にクリアされたときにアボートが生成されます。
- CPSR.A ビットが 1 の場合、データアボート例外が生成されます。*v6 デバッグのデバッグ状態での未定義命令およびデータアボート例外* P. C5-23 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.7.1 Undefined Instruction and Data Abort exceptions in Debug state in v6 Debug

In v6 Debug, if an Undefined Instruction exception is generated when the processor is in Jazelle state and Debug state, the result is UNPREDICTABLE.

Otherwise, in v6 Debug, Undefined Instruction and Data Abort exceptions generated in Debug state are taken by the processor as follows:

- The PC, CPSR, and SPSR_<exception_mode> are set in the same way as in a normal Non-debug state exception entry. In addition:
  - — if the exception is an asynchronous abort, and the PC has not yet been written, LR_abt is set as for exception entry in Non-debug state
  - — in all other cases, LR_<exception_mode> is set to an UNKNOWN value.
- The processor remains in Debug state, and does not prefetch the exception vector.

In addition, for a Data Abort exception:

- The DFSR and DFAR are set in the same way as in a normal Non-debug state exception entry. The DBGWFAR is set to an UNKNOWN value. The IFSR is not modified.
- DBGDSCR.ADABORT_l or DBGDSCR.SDABORT_l is set to 1.
- The DBGDSCR.MOE bits are set to 0b0110, D-side abort occurred.

For more information about asynchronous aborts in ARMv6 see *Behavior in ARMv6* on page C5-6.

Debuggers must take care when processing a debug event that occurred when the processor was executing an exception handler. The debugger must save the values of SPSR_und and LR_und before performing any operation that might result in an Undefined Instruction exception being generated in Debug state. The debugger must also save the values of SPSR_abt and LR_abt, and of the DFSR, DFAR and DBGWFAR before performing an operation that might generate a Data Abort exception when in Debug state. If this is not done, register values might be overwritten, resulting in UNPREDICTABLE software behavior.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.7.1 v6 デバッグのデバッグ状態での未定義命令およびデータアボート例外

v6 デバッグでは、プロセッサが Jazelle 状態かつデバッグ状態のときに未定義命令例外が生成されると、結果は予測不能です。

それ以外の場合、v6 デバッグでは、デバッグ状態で生成された未定義命令およびデータアボート例外は、プロセッサにより次のように取得されます。

- PC、CPSR、SPSR_< 例外モード > は、通常の非デバッグ状態での例外開始時と同様に設定されます。さらに、次の動作が行われます。
  - – 例外が非同期アボートで、PC がまだ書き込まれていない場合、LR_abt は非デバッグ状態での例外開始時と同様に設定されます。
  - – 他のすべての場合、LR_< 例外モード > に設定される値は未知です。
- プロセッサはデバッグ状態のままで、例外ベクタをプリフェッチしません。

これらに加えて、データアボート例外については次の動作が行われます。

- DFSR と DFAR は、通常の非デバッグ状態での例外開始時と同様に設定されます。DBGWFAR に設定される値は未知です。IFSR は変更されません。
- DBGDSCR.ADABORT_l または DBGDSCR.SDABORT_l が 1 にセットされます。
- DBGDSCR.MOE ビットは 0b0110 に設定され、D 側アボートが発生したことを示します。

ARMv6 での非同期アボートの詳細については、*ARMv6 での動作* P. C5-6 を参照して下さい。

プロセッサが例外ハンドラを実行中に発生したデバッグイベントを、デバッガで処理するときには注意が必要です。デバッガは、デバッグ状態で未定義命令例外が生成される可能性がある動作を実行する前に、SPSR_und および LR_und の値を必ず保存する必要があります。また、デバッグ状態でデータアボート例外を生成する可能性がある動作を実行する前に、SPSR_abt、LR_abt、DFSR、DFAR、DBGWFAR の値を必ず保存する必要があります。この操作を行わない場合、レジスタの値が上書きされ、ソフトウェアの予測不能な動作を引き起こすことがあります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.8 Memory system behavior in Debug state

The Debug architecture places requirements on the memory system. There are two general guidelines:

- Memory coherency has to be maintained during debugging.
- It is best if debugging is non-intrusive. This requires a way to preserve, for example, the contents of memory caches and translation lookaside buffers (TLBs), so the state of the target application is not altered.

In Debug state, it is strongly recommended that the caches and TLBs, where implemented, behave as described here. For preservation purposes it is strongly recommended that it is possible to:

- disable cache evictions and linefills, so that cache accesses, on read or write, do not cause the contents of caches to change.
- disable TLB evictions and replacements, so that translations do not cause the contents of TLBs to change.

The mechanisms for disabling these operations:

- must be accessible by the external debugger
- are only required when in Debug state.

In v6.1 Debug and v7 Debug, the Debug State Cache Control Register (DBGDSCCR) and the Debug State MMU Control Register (DBGDSMCR) are used for this purpose.

While the processor is in Debug state, no instruction fetches occur and therefore:

- if the system implements separate instruction and data caches then there might be no instruction cache evictions or replacements
- if the system implements separate instruction and data TLBs then there might be no instruction TLB evictions or replacements.

In Debug state, reads must behave as in Non-debug state:

- cache reads return data from the cache
- cache misses fetch from external memory.

A debugger must be able to maintain coherency between instruction and data memory, and maintain coherency in a multiprocessor system. This means that in Debug state a debugger must be able to force all writes to update all levels of memory to the point of coherency.

It must be possible to reset the memory system of the processor to a known safe and coherent state. Also, it must be possible to reset any caches of meta-information, such as branch predictor arrays, to a safe and coherent state.

For debugging purposes ARM recommends that TLBs can be disabled so that all TLB accesses are read from the main translation tables, and not from the TLB. This enables a debugger to access memory without using any virtual to physical memory mapping that is implemented for the application.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C5.8 デバッグ状態でのメモリシステムの動作

デバッグアーキテクチャでは、メモリシステムに関する必要条件があります。一般的なガイドラインは 2 つです。

- デバッグ中にメモリコヒーレンシが維持される必要があります。
- デバックが非侵襲性であることが最適です。このためには、例えばメモリキャッシュと変換ルックアサイドバッファ (TLB) の内容を保持し、ターゲットアプリケーションの状態が変更されないようにする方法が必要です。

デバッグ状態では、キャッシュと TLB が実装されている場合、ここで説明するように動作することを強く推奨します。保持の目的から、次の動作が可能であることが強く推奨されます。

- キャッシュの退出とラインフィルを不可能にし、読み出し / 書き込みのキャッシュアクセスによってキャッシュの内容が変更されないようにする。
- TLB 退出と置き換えを不可能にし、変換によって TLB の内容が変更されないようにする。

これらの動作を不可能にするための機構には、次の条件があります。

- 外部デバッガによりアクセス可能であること。
- デバッグ状態でのみ必要である。

v6.1 デバッグおよび v7 デバッグでは、デバッグ状態キャッシュ制御レジスタ (DBGDSCCR) とデバッグ状態 MMU 制御レジスタ (DBGDSMCR) がこの目的に使用されます。

プロセッサがデバッグ状態の間は命令フェッチが発生しないため、次の条件が適用されます。

- システムに独立した命令キャッシュとデータキャッシュが実装されている場合、命令キャッシュの退出や置き換えが行われない可能性があります。
- システムに独立した命令 TLB とデータ TLB が実装されている場合、命令 TLB の退出や置き換えが行われない可能性があります。

デバッグ状態では、読み出しは次のように、非デバッグ状態と同様に動作する必要があります。

- キャッシュ読み出しでは、キャッシュからデータが返される。
- キャッシュミスの場合、外部メモリからフェッチが行われる。

デバッガは、命令およびデータメモリの間でコヒーレンシを維持することができる必要があり、マルチプロセッサシステムでもコヒーレンシを維持する必要があります。つまり、デバッグ状態では、デバッガはすべての書き込みについて、すべてのレベルのメモリがコヒーレンシ時点に達するまで更新を強制することが可能な必要があります。

プロセッサのメモリシステムを、既知の安全でコヒーレントな状態にリセットすることが可能な必要があります。また、分岐予測器アレイなど、メタ情報のキャッシュをすべて、安全でコヒーレントな状態にリセットすることが可能な必要があります。

デバッグの目的では、TLB を非稼動にし、すべての TLB アクセスが TLB からではなくメイン変換テーブルから読み出されるようにすることを可能にすることをお勧めします。これによって、デバッガはアプリケーション用に実装されている、仮想メモリから物理メモリへのマッピングを使用せずに、メモリにアクセスできます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.8.1 Access to specific cache management functions in Debug state

If a processor includes the Security Extensions and supports Secure User halting debug, it must implement mechanisms that enable memory system requirements to be met when debugging in Secure User mode when invasive debug is not permitted in Secure privileged modes. This is a situation where executing the CP15 cache and TLB control operations would otherwise be prohibited.

To meet these requirements, ARM recommends that, on a processor that implements the Security Extensions and supports Secure User halting debug, when the processor is in Debug state:

- the rules for accessing CP15 registers do not apply for a certain set of register access operations
- the set of operations depends on the Debug architecture version, as shown in Table C5-5.

Table C5-5 CP15 operations permitted from User mode in Debug state

Versions	Operation	Description
v7 Debug	`MCR p15,0,<Rt>,c7,c5,0`	Invalidate entire instruction cache and flush branch predictor arrays [a]
	`MCR p15,0,<Rt>,c7,c5,1`	Invalidate instruction cache by MVA [a]
	`MCR p15,0,<Rt>,c7,c5,7`	Invalidate MVA from branch predictor array
	`MCR p15,0,<Rt>,c7,c10,1`	Clean data or unified cache line by MVA to point of coherency [b]
	`MCR p15,0,<Rt>,c7,c10,2`	Clean data or unified cache line by set/way [b]
	`MCR p15,0,<Rt>,c7,c11,1`	Clean data or unified cache line by MVA to point of unification [b]
	`MCR p15,0,<Rt>,c7,c1,0`	Invalidate entire instruction cache Inner Shareable [c]
	`MCR p15,0,<Rt>,c7,c1,6`	Invalidate entire branch predictor array Inner Shareable [c]
v6.1 Debug	`MCRR p15,0,<Rt>,<Rn>,c5`	Invalidate instruction cache by VA range
v6.1 Debug, v7 Debug	`MCR p15,0,<Rt>,c7,c5,6`	Flush entire branch predictor array

a. See also *v7 Debug restrictions on instruction cache invalidation in Secure User debug* on page C5-26.
b. A debugger does not have to perform cache cleaning operations if DBGDSCCR.nWT is implemented and is set to 0, see *Debug State Cache Control Register (DBGDSCCR)* on page C10-81. This is because when nWT is set to 0, writes do not leave dirty data in the cache that is not coherent with outer levels of memory. However, the I-cache is not updated, so I-cache invalidate operations are required.
c. These instructions are part of the Multiprocessing Extensions. See *Multiprocessor effects on cache maintenance operations* on page B2-23.

These instructions must be executable in Debug state regardless of any processor setting. However, use of an operation can generate an abort if instruction cache lockdown is in use.

For more information about debug access to coprocessor instructions, see *Coprocessor and Advanced SIMD instructions in Debug state* on page C5-16.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.8.1 デバッグ状態での特定のキャッシュ管理機能へのアクセス

プロセッサにセキュリティ拡張機能が含まれ、セキュアユーザ ホールトデバッグがサポートされている場合、セキュア特権モードで侵襲性デバッグが許可されてないときのセキュアユーザモードでデバッグを行うときに、メモリシステムの要件を満たすことを可能にする機構を実装する必要があります。これは、そうでない場合には CP15 キャッシュおよび TLB の制御操作の実行が禁止される状況です。

これらの要件を満たすため、セキュリティ拡張機能を実装し、セキュアユーザ ホールトデバッグをサポートしているプロセッサでは、プロセッサがデバッグ状態のときに次の条件を満たすことを推奨します。

- CP15 レジスタにアクセスするための規則が、特定のレジスタアクセス操作の組には適用されない。
- この操作の組は、表 C5-5 に示すように、デバッグアーキテクチャのバージョンによって異なります。

表 C5-5 デバッグ状態のユーザモードから許可される CP15 操作

バージョン	操作	説明
v7 デバッグ	`MCR p15,0,<Rt>,c7,c5,0`	命令キャッシュ全体を無効化し、分岐予測器アレイをフラッシュします。[a]
	`MCR p15,0,<Rt>,c7,c5,1`	MVA により命令キャッシュを無効化します。[a]
	`MCR p15,0,<Rt>,c7,c5,7`	分岐予測器アレイから MVA を無効化します。
	`MCR p15,0,<Rt>,c7,c10,1`	MVA により、データまたは統一キャッシュラインを、コヒーレンシ時点に達するまでクリーニングします。[b]
	`MCR p15,0,<Rt>,c7,c10,2`	セット / ウェイによりデータまたは統一キャッシュラインをクリーニングします。[b]
	`MCR p15,0,<Rt>,c7,c11,1`	MVA により、データまたは統一キャッシュラインを、統一時点に達するまでクリーニングします。[b]
	`MCR p15,0,<Rt>,c7,c1,0`	命令キャッシュ全体の内部共有可能を無効化します。[c]
	`MCR p15,0,<Rt>,c7,c1,6`	分岐予測器アレイ全体の内部共有可能を無効化します。[c]
v6.1 デバッグ	`MCRR p15,0,<Rt>,<Rn>,c5`	VA 範囲により命令キャッシュを無効化します。
v6.1 デバッグ、v7 デバッグ	`MCR p15,0,<Rt>,c7,c5,6`	分岐予測器アレイ全体をフラッシュします。

a. *セキュアユーザ デバッグでの命令キャッシュ無効化に関する v7 デバッグでの制限* P. C5-26 も参照して下さい。
b. DBGDSCCR.nWT が実装され、0 に設定されている場合、デバッガがキャッシュのクリーニング操作を実行する必要はありません。*デバッグ状態キャッシュ制御レジスタ (DBGDSCCR)* P. C10-81 を参照して下さい。これは、nWT が 0 に設定されている場合、書き込みによって、メモリの外部レベルとコヒーレントでないダーティデータがキャッシュに残されることはないためです。ただし、I キャッシュは更新されないため、I キャッシュの無効化操作は必要です。
c. これらの命令はマルチプロセッシング拡張機能の一部です。*キャッシュの保守操作に対するマルチプロセッサの影響* P. B2-23 を参照して下さい。

これらの命令は、プロセッサのどの設定にもかかわらず、デバッグ状態で実行可能な必要があります。ただし、命令キャッシュロックダウンが使用されている場合、この操作を使用するとアボートが生成されることがあります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For more information about the ARMv7 cache maintenance operations, see:

- *CP15 c7, Cache and branch predictor maintenance functions* on page B3-126 for a VMSA implementation
- *CP15 c7, Cache and branch predictor maintenance functions* on page B4-68 for a PMSA implementation.

In v6 Debug and on any processor that does not implement Security Extensions, or when debugging in a state and mode where privileged CP15 operations can be executed, the debugger can use any CP15 operations. These include, but are not limited to, those operations listed in Table C5-5 on page C5-25.

v7 Debug restrictions on instruction cache invalidation in Secure User debug

An ARMv7 implementation that includes the Security Extensions and supports Secure User halting debug must support Secure User debug access to at least one of these instruction cache invalidation operations:

- Invalidate entire instruction cache, and flush branch predictor arrays, `MCR p15,0,<Rt>,c7,c5,0`
- Invalidate instruction cache by MVA, `MCR p15,0,<Rt>,c7,c5,1`.

An implementation might support both of these operations.

If the DSCCR.nWT bit is not implemented, the implementation must also support Secure User debug access to at least the operation to Clean data or unified cache line by MVA to point of coherency.

A debugger requires access to an instruction cache invalidation operations so that it can maintain coherency between instruction memory and data memory, and between processors in a multiprocessor system.

However, the architecture imposes restrictions on the operation of these instructions in Debug state, that are not required when the instructions are used in normal operation. In Secure User mode in Debug state when invasive debug is not permitted in Secure privileged modes:

- If the *Invalidate all instruction caches* operation is supported it must:
  - — invalidate all unlocked lines in the cache
  - — leave any locked lines in the cache unchanged.

  If there are locked lines in the cache the instruction can abort, but only after it has invalidated all unlocked lines. However, there is no requirement for the operation to abort if there are locked lines.
- If the *Invalidate instruction caches by MVA* operation is supported, this operation must not invalidate a locked line. If an instruction attempts to invalidate a locked line in Secure User mode debug the implementation must either:
  - — ignore the instruction
  - — abort the instruction.

These requirements mean that these instructions might operate differently in Debug state to how they operate in Non-debug state.

Note

In ARMv7, it is IMPLEMENTATION DEFINED whether instruction cache locking is supported.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

コプロセッサ命令へのデバッグアクセスの詳細については、*デバッグ状態でのコプロセッサおよびアドバンスト SIMD 命令* P. C5-16 を参照して下さい。

ARMv7 のキャッシュ保守操作の詳細については、次に示すセクションを参照して下さい。

- *CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126（VMSA 実装の場合）
- *CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68 （PMSA 実装の場合）

v6 デバッグで、セキュリティ拡張機能を実装していないプロセッサの場合、または特権 CP15 操作が実行可能な状態およびモードでデバッグを行う場合は、デバッガは任意の CP15 操作を使用できます。これには、P. C5-25 の表 C5-5 に一覧表記されている操作を含みますが、これだけには限りません。

セキュアユーザ デバッグでの命令キャッシュ無効化に関する v7 デバッグでの制限

セキュリティ拡張機能を含み、セキュアユーザ ホールトデバッグをサポートする ARMv7 実装では、次に示す命令キャッシュ無効化操作のうち最低 1 つに対して、セキュアユーザ デバッグでのアクセスをサポートする必要があります。

- 命令キャッシュ全体を無効化し、分岐予測器アレイをフラッシュする。`MCR p15,0,<Rt>,c7,c5,0`
- MVA により命令キャッシュを無効化する。`MCR p15,0,<Rt>,c7,c5,1`

実装でこれらの操作の両方をサポートすることもできます。

DSCCR.nWT ビットが実装されていない場合、実装では最低でも、MVA によってデータまたは統一キャッシュラインを、コヒーレンシ時点に達するまでクリーニングする操作に対しても、セキュアユーザ デバッグをサポートする必要があります。

デバッガは、命令メモリとデータメモリとの間、およびマルチプロセッサシステムのプロセッサ間でコヒーレンシを維持するため、命令キャッシュの無効化操作にアクセスする必要があります。

ただし、デバッグ状態でのこれらの命令の動作には、アーキテクチャにより、通常動作でこれらの命令が使用されるときには必要のない制限が指定されます。セキュア特権モードで侵襲性デバッグが許可されていないとき、デバッグ状態のセキュアユーザ モードでは次の条件が適用されます。

- *すべての命令キャッシュを無効化する*操作がサポートされている場合、必ず次の動作を行う必要があります。
  - – キャッシュ内のロックされていないすべてのラインを無効化する。
  - – キャッシュ内のロックされているラインは一切変更しない。

  キャッシュにロックされているラインが存在する場合、この命令でアボートが発生してもかまいません。ただし、ロックされていないすべてのラインが無効化された後でアボートが発生する必要があります。ただし、ロックされているラインが存在する場合に、操作でアボートが発生することは必須ではありません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.8.2 Debug state Cache and MMU Control Registers

In v6 Debug, the Debug state MMU Control Register (DBGDSMCR) and Debug state Cache Control Register (DBGDSCCR) are not defined.

In v6.1 Debug, ARM recommends the debug registers DBGDSMCR and DBGDSCCR.

v7 Debug requires DBGDSMCR and DBGDSCCR, but there can be IMPLEMENTATION DEFINED limits on their behavior.

For descriptions of these registers, see *Memory system control registers* on page C10-80.

In all debug implementations there can be IMPLEMENTATION DEFINED support for cache behavior override and, on a VMSA implementation, for TLB debug control.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- *MVA により命令キャッシュを無効化する*操作がサポートされている場合、この操作はロックされているラインを無効化することはできません。セキュアユーザ モードのデバッグで、命令がロックされているラインの無効化を試みた場合、実装は次のいずれかの動作を行う必要があります。
  - 命令を無視する。
  - 命令でアボートを生成する。

これらの条件は、これらの命令がデバッグ状態では、非デバッグ状態とは異なる動作を行う可能性があることを意味します。

注

ARMv7 では、命令キャッシュのロックをサポートするかどうかは実装定義です。

C5.8.2 デバッグ状態キャッシュおよび MMU 制御レジスタ

v6 デバッグでは、デバッグ状態 MMU 制御レジスタ (DBGDSMCR) およびデバッグ状態キャッシュ制御レジスタ (DBGDSCCR) は定義されていません。

v6.1 デバッグでは、デバッグレジスタ DBGDSMCR および DBGDSCCR が推奨されています。

v7 デバッグでは DBGDSMCR および DBGDSCCR が必須ですが、これらのレジスタの動作に実装定義の制限を設けることができます。

これらのレジスタの説明については、*メモリシステム制御レジスタ* P. C10- 80 を参照して下さい。

すべてのデバッグ実装について、キャッシュ動作のオーバライドを実装定義でサポートできます。また、VMSA 実装については、TLB デバッグ制御についても同様のサポートを行うことができます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C5.9 Leaving Debug state

The processor leaves Debug state when a restart request command is received. A restart request can be one of the following:

- An External Restart request. This is a request from the system for the processor to leave Debug state. The External Restart request enables multiple processors to be restarted synchronously.

  The External Restart request is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an External Restart request input to the processor.

- A restart request command.

  In v7 Debug, the restart request command is made by a debugger writing 1 to the DBGDRCR Restart request bit, see *Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29

A number of flags in the *Debug Status and Control Register* (DBGDSCR) must be set correctly before leaving Debug state, see *Debug Status and Control Register (DBGDSCR)* on page C10-10. The flags that must be set are:

- the sticky exception flags, DBGDSCR[8:6], must be set to 0b000
- the Execute ARM Instruction Enable bit, DBGDSCR.ITRen, must be set to 0
- the Latched Instruction Complete flag, DBGDSCR.InstrCompl_l, must be set to 1.

In v7 Debug the sticky exception flags are cleared to 0 by writing 1 to the Clear Sticky Exceptions bit of the DBGDRCR. This operation can be combined with the restart request command. For more information see *Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29.

If the processor is signaled to leave Debug state without all of these flags set to the correct values the results are UNPREDICTABLE.

On receipt of a restart request, the processor performs a sequence of operations to leave Debug state.

If DBGDSCR is read during the restart sequence, DBGDSCR.RESTARTED must read as 0 and DBGDSCR.HALTED must read as 1. At all other times DBGDSCR.RESTARTED must read as 1.

On completion of the restart sequence, the processor leaves Debug state:

- DBGDSCR.HALTED is set to 0.
- The processor stops ignoring debug events and starts executing instructions from the address held in the PC, in the mode and instruction set state indicated by the current value of the CPSR. The execution state bits of the CPSR are honored, and the IT bits state machine is restarted, with the current value applying to the first instruction executed.
- Unless the DBGDSCR.DBGack bit is set to 1, the processor signals to the system that it is in Non-debug state. Details of this signalling method, including whether it is implemented, are IMPLEMENTATION DEFINED.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C5.9 デバッグ状態の終了

プロセッサは、再始動要求コマンドを受信したときにデバッグ状態を終了します。再始動要求は次のいずれかです。

- 外部からの再始動要求。これは、プロセッサがデバッグ状態を終了することをシステムから要求するものです。外部からの再始動要求により、複数のプロセッサが同期して再始動することが可能になります。

  外部からの再始動要求は、実装定義の手段によって生成されます。通常は、プロセッサへの外部再始動要求入力をアサートすることによって生成されます。

- 再始動要求コマンド。v7 デバッグでは、再始動要求コマンドは、デバッガが DBGDRCR の再始動要求ビットに 1 を書き込むことで行われます。*デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10-29 を参照して下さい。

デバッグ状態を終了する前に、*デバッグステータスおよび制御レジスタ* (DBGDSCR) の多くのフラグを正しく設定する必要があります。*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。設定の必要があるフラグは次に示すものです。

- スティッキー例外フラグ DBGDSCR[8:6] を 0b000 に設定する必要があります。
- ARM 命令実行イネーブルビット (DBGDSCR.ITRen) を 0 にセットする必要があります。
- ラッチ命令比較フラグ (DBGDSCR.InstrCompl_l) を 1 にセットする必要があります。

v7 デバッグでは、DBGDRCR のスティッキー例外クリアビットに 1 を書き込むと、スティッキー例外フラグが 0 にクリアされます。この操作は、再始動要求コマンドと組み合わせ可能です。詳細については、*デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10-29 を参照して下さい。

これらのフラグすべてを正しい値に設定せず、プロセッサに対してデバッグ状態を終了するよう通知した場合、結果は予測不能です。

プロセッサは、再始動要求を受信すると、デバッグ状態を終了するための動作シーケンスを実行します。

再始動シーケンスの間に DBGDSCR が読み出された場合、DBGDSCR.RESTARTED は 0、DBGDSCR.HALTED は 1 として読み出される必要があります。他のすべての時点では、DBGDSCR.RESTARTED は 1 として読み出される必要があります。

再始動シーケンスの完了時に、プロセッサはデバッグ状態を終了します。

- DBGDSCR.HALTED が 0 にセットされます。
- プロセッサは、デバッグイベントの無視を中止し、CPSR の現在の値によって示されるモードおよび命令セット状態で、PC に保持されているアドレスから命令の実行を開始します。CPSR の実行状態ビットは維持され、IT ビット ステートマシンが再始動され、実行される最初の命令に現在の値が適用されます。
- DBGDSCR.DBGack ビットが 1 にセットされていない限り、プロセッサはシステムに対して、非デバッグ状態であることを通知します。この通知の送信方法の詳細は、それを実装するかどうかも含めて実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

Leaving Debug state is not a memory barrier operation. This means that:

- If a debugger executes any context altering operations in Debug state, it must issue an Instruction Synchronization Barrier (ISB) instruction before leaving Debug state
- If the debugger executes any memory access instructions in Debug state, it must execute a Data Synchronization Barrier (DSB) instruction before leaving Debug state, to ensure those accesses are complete. This DSB might form part of the IMPLEMENTATION DEFINED sequence of instructions required to ensure that the processor has recognized any asynchronous aborts, as described in *Asynchronous aborts and entry to Debug state* on page C5-5.

For details of the recommended external debug interface, see *Run-control and cross-triggering signals* on page AppxA-5 and *DBGACK and DBGCPUDONE* on page AppxA-7.

In v6 Debug and v6.1 Debug:

- the DBGDRCR and External Restart request are not supported
- if the processor implements the recommended ARM Debug Interface v4, the restart request command is issued through the JTAG interface by placing the RESTART instruction in the IR and taking the Debug TAP State Machine through the Run-Test/Idle state. Connecting multiple JTAG interfaces in series enables multiple processors to be restarted synchronously.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

デバッグ状態の終了はメモリバリア操作ではありません。これは次のことを意味します。

- デバッガが、デバッグ状態でコンテキスト変更操作を実行した場合、デバッグ状態を終了する前に命令同期バリア (ISB) 命令を発行する必要があります。
- デバッガが、デバッグ状態でメモリアクセス命令を実行した場合、それらのアクセスが完了したことを保証するため、デバッグ状態を終了する前にデータ同期バリア (DSB) 命令を実行する必要があります。*非同期アボートとデバッグ状態への遷移* P. C5-5 で説明されているように、DSB は、プロセッサがすべての非同期アボートを認識したことを保証するために必要となる、実装定義の命令シーケンスの一部となる場合があります。

推奨される外部デバッグインタフェースの詳細については、*実行制御およびクロストリガ信号* P. 付録 A-5 および *DBGACK および DBGCPUDONE* P. 付録 A-7 を参照して下さい。

v6 デバッグおよび v6.1 デバッグでは、次の条件が適用されます。

- DBGDRCR および外部からの再始動要求はサポートされません。
- プロセッサに、推奨の ARM デバッグインタフェース v4 が実装されている場合、再始動要求コマンドは JTAG インタフェースから、IR に RESTART 命令を置き、実行 - テスト / アイドル状態を経由してデバッグ TAP ステートマシンを取得することにより発行されます。複数の JTAG インタフェースを直列に接続すると、複数のプロセッサを同期して再始動することができます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter C6
Debug Register Interfaces

This chapter describes the debug register interfaces. It contains the following sections:

- *About the debug register interfaces* on page C6-2
- *Reset and power-down support* on page C6-4
- *Debug register map* on page C6-18
- *Synchronization of debug register updates* on page C6-24
- *Access permissions* on page C6-26
- *The CP14 debug register interfaces* on page C6-32
- *The memory-mapped and recommended external debug interfaces* on page C6-43.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 C6 章
デバッグレジスタ インタフェース

本章では、デバッグレジスタ インタフェースについて説明します。本章は次のセクションから構成されています。

- *デバッグレジスタ インタフェースについて* P. C6-2
- *リセットと電力オフのサポート* P. C6-4
- *デバッグレジスタのマップ* P. C6-18
- *デバッグレジスタの更新の同期* P. C6-23
- *アクセス許可* P. C6-25
- *CP14 デバッグレジスタ インタフェース* P. C6-32
- *メモリマップされたインタフェースと推奨される外部デバッグインタフェース* P. C6-43

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.1 About the debug register interfaces

The Debug architecture defines a set of debug registers. The debug register interfaces provide access to these registers. This chapter describes the different ways of implementing the debug register interfaces.

The debug register interfaces provide access to the debug registers from:

- software running on the processor, see *Processor interface to the debug registers*
- an external debugger, see *External interface to the debug registers*.

The debug register interfaces always include the *Debug Communications Channel*, see *The Debug Communications Channel (DCC)* on page C6-3.

C6.1.1 Processor interface to the debug registers

Table C6-4 on page C6-32 lists the set of CP14 debug instructions for accessing the debug registers that must be implemented.

The possible interfaces between the software running on the processor and the debug registers are:

- The Baseline CP14 interface. This provides access to a small set of the debug registers through a set of coprocessor instructions. It must be implemented by all processors.
- The Extended CP14 interface. This provides access to the remaining debug registers through a coprocessor interface. It is required in v6 Debug and v6.1 Debug, and is optional in v7 Debug.
- The memory-mapped interface. This provides memory-mapped access to the debug registers. It is introduced in v7 Debug, and is an optional interface. When it is implemented:
  - — some of the registers that are accessed through the Baseline CP14 interface are not available through the memory-mapped interface
  - — it is IMPLEMENTATION DEFINED whether the memory-mapped interface is visible only to the processor in which the debug registers are implemented, or is also visible to other processors in the system.

An ARMv7 implementation must include the Baseline CP14 interface and at least one of:

- the Extended CP14 interface
- the memory-mapped interface.

C6.1.2 External interface to the debug registers

Every ARMv6 and ARMv7 implementation must include an external debug interface. This interface provides access to the debug registers from an external debugger through a *Debug Access Port* (DAP). This interface is IMPLEMENTATION DEFINED. For details of the interface recommended by ARM:

- for an ARMv7 implementation, see the *ARM Debug Interface v5 Architecture Specification*
- for an ARMv6 implementation, contact ARM.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C6.1 デバッグレジスタ インタフェースについて

デバッグアーキテクチャでは、一連のデバッグレジスタが定義されています。デバッグレジスタ インタフェースは、これらのレジスタへのアクセスを提供します。本章では、デバッグレジスタ インタフェースを実装する各種の方法について説明します。

デバッグレジスタ インタフェースは、次のものからデバッグレジスタにアクセスするために使用されます。

- プロセッサで実行されているソフトウェア。*プロセッサからデバッグレジスタへのインタフェース*を参照して下さい。
- 外部デバッガ。*デバッグレジスタへの外部インタフェース*を参照して下さい。

デバッグレジスタ インタフェースには、*デバッグ通信チャネル*が必ず含まれています。*デバッグ通信チャネル (DCC)* P. C6-3 を参照して下さい。

C6.1.1 プロセッサからデバッグレジスタへのインタフェース

デバッグレジスタにアクセスするため、必ず実装しなければならない CP14 デバッグ命令の一覧を、P. C6-32 の表 C6-4 に示します。

プロセッサで実行されているソフトウェアと、デバッグレジスタとの間のインタフェースには、次のものが使用できます。

- ベースライン CP14 インタフェース。このインタフェースでは、一連のコプロセッサ命令を使用して、デバッグレジスタの小さな組にアクセスできます。すべてのプロセッサで実装する必要があります。
- 拡張 CP14 インタフェース。コプロセッサインタフェース経由で、残りのデバッグレジスタへのアクセスを提供します。v6 デバッグおよび v6.1 デバッグでは必須、v7 デバッグではオプションです。
- メモリマップされたインタフェース。デバッグレジスタへのメモリマップによるアクセスを提供します。これは v7 デバッグで新たに導入されたもので、オプションのインタフェースです。実装する場合、次の条件が適用されます。
  - ベースライン CP14 インタフェース経由でアクセスされるレジスタのいくつかは、メモリマップされたインタフェースではアクセスできません。
  - メモリマップされたインタフェースが、デバッグレジスタを実装しているプロセッサからのみ可視であるか、システム内の他のプロセッサからも可視であるかは実装定義です。

ARMv7 実装には、ベースライン CP14 インタフェースと、次のうち最低 1 つが含まれている必要があります。

- 拡張 CP14 インタフェース
- メモリマップされたインタフェース

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The Debug architecture does not require implementation of the recommended interface. However:

- the ARM RealView tools require the recommended interface
- ARM recommends this interface for compatibility with other tool chains.

C6.1.3 The Debug Communications Channel (DCC)

The debug register interface includes the *Debug Communications Channel* (DCC). This is accessed through two physical registers:

- DBGDTRTX, for data transfers from the processor to an external debugger
- DBGDTRRX, for data transfers from the external debugger to the processor.

In addition, there are four DCC status flags in the DBGDSCR:

- TXfull and TXfull_l, indicating the DBGDTRTX status
- RXfull and RXfull_l, indicating the DBGDTRRX status.

There are separate internal and external views of the DBGDSCR, and of the DBGDTRTX and DBGDTRRX Registers:

- DBGDTRTXint, DBGDTRRXint and DBGDSCRint provide the internal view
- DBGDTRTXext, DBGDTRRXext and DBGDSCRext provide the external view.

For more information, see *Internal and external views of the DBGDSCR and the DCC registers* on page C6-21.

Note

In previous descriptions of the DCC, the term DTR (*Data Transfer Register*) is used to describe the DCC data registers. In those descriptions, the DBGDTRTX Register is named wDTR, and the DBGDTRRX Register is named rDTR.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.1.2 デバッグレジスタへの外部インタフェース

すべての ARMv6 および ARMv7 実装には、外部デバッグインタフェースが含まれている必要があります。このインタフェースは、外部のデバッガからデバッグレジスタへ、*デバッグアクセスポート* (DAP) 経由でのアクセスを提供します。このインタフェースは実装定義です。推奨されるインタフェースの詳細については、次を参照して下さい。

- ARMv7 実装については、『*ARM デバッグインタフェース v5、アーキテクチャ仕様書*』を参照して下さい。
- ARMv6 実装については、ARM にお問い合わせ下さい。

デバッグアーキテクチャでは、推奨のインタフェースを実装することは必須ではありません。ただし、以下のような制限があります。

- ARM RealView ツールには、推奨のインタフェースが必要です。
- 他のツールチェインとの互換性のため、このインタフェースを推奨します。

C6.1.3 デバッグ通信チャネル (DCC)

デバッグレジスタ インタフェースには、*デバッグ通信チャネル* (DCC) が含まれています。このチャネルには、次の 2 つの物理レジスタ経由でアクセスします。

- プロセッサから外部デバッガへのデータ転送には、DBGDTRTX を使用します。
- 外部デバッガからプロセッサへのデータ転送には、DBGDTRRX を使用します。

さらに、DBGDSCR には次の 4 つの DCC ステータスフラグがあります。

- TXfull および TXfull_l。DBGDTRTX のステータスを示します。
- RXfull および RXfull_l。DBGDTRRX のステータスを示します。

DBGDSCR、および DBGDTRTX と DBGDTRRX レジスタには、独立した内部および外部のビューがあります。

- DBGDTRTXint、DBGDTRRXint、DBGDSCRint は内部ビューを提供します。
- DBGDTRTXext、DBGDTRRXext、DBGDSCRext は外部ビューを提供します。

詳細については、*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6- 21 を参照して下さい。

注

DCC の以前の説明では、DTR（*データ転送レジスタ*）という用語は、DCC データレジスタを示すために使用されていました。これらの説明では、DBGDTRTX レジスタは wDTR、DBGDTRRX レジスタは rDTR と呼ばれています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.2 Reset and power-down support

This section contains the following subsections:

- *Debug guidelines for systems with energy management capability*
- *Power domains and debug* on page C6-5
- *The OS Save and Restore mechanism* on page C6-8
- *Recommended reset scheme for v7 Debug* on page C6-16.

C6.2.1 Debug guidelines for systems with energy management capability

ARMv7 processors can be built with energy management capabilities. This section describes how to use the v7 Debug features to debug software running on these systems.

v7 Debug only permits debugging software that is running on a system where:

- energy-saving measures are taken only when the processor is in an idle state
- it is a function of the operating system, or other supervisor code, to take any implemented energy-saving measures.

The measures that the OS can take to save energy during an idle state can be split in two groups:

Standby The OS takes some measures, including using IMPLEMENTATION DEFINED features, to reduce energy consumption. The processor preserves the processor state, including the debug logic state. Changing from standby to normal operation does not involve a reset of the processor.

Power-down The OS takes some measures to reduce energy consumption. These measures mean the processor cannot preserve the processor state, and therefore the measures taken must include the OS saving any processor state it requires not to be lost. Changing from power-down to normal operation must include:

- a reset of the processor, after the power level has been restored
- reinstallation of the processor state by the OS.

Standby is the least invasive OS energy saving state. It implies only that the processor is unavailable, and does not clear any of the debug settings. For standby, v7 Debug prescribes only the following:

- If the processor is in standby and a Halting debug event is triggered the processor must leave standby to handle the debug event. If the processor executed a `WFI` or `WFE` instruction to enter standby then that instruction is retired.
- If the processor is in standby and the external debug or memory-mapped interface is accessed, the processor must automatically:
  - — leave standby
  - — respond to the debug transaction
  - — go back to standby.

  This is possible because the external debug and memory-mapped interface can insert wait states, for example by holding **PREADYDBG** LOW, until the processor has left standby.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C6.2 リセットと電力オフのサポート

このセクションは、次のサブセクションから構成されています。

- *電力管理機能を持つシステムでのデバッグのガイドライン*
- *電力ドメインとデバッグ* P. C6-5
- *OS 保存および復元機構* P. C6-8
- *v7 デバッグで推奨されるリセット方式* P. C6-16

C6.2.1 電力管理機能を持つシステムでのデバッグのガイドライン

ARMv7 プロセッサには、電力管理機能を組み入れることができます。このセクションでは、これらのシステムで実行されているソフトウェアに対して v7 デバッグ機能を使用する方法について説明します。

v7 デバッグでは、システムで実行されているソフトウェアについて、次の条件が当てはまる場合のみデバッグが許可されます。

- プロセッサがアイドル状態の場合のみ、電力削減手法が適用される。
- 実装されている電力削減手法を適用するのが、必ずオペレーティングシステムまたは他のスーパバイザコードの機能である。

アイドル状態での電力削減のために OS が使用できる手法は、2 つのグループに分けられます。

スタンバイ OS が、実装定義の機能の使用を含むいくつかの手法により、電力消費を削減します。プロセッサは、デバッグロジックの状態を含むプロセッサ状態を保持します。スタンバイから通常の動作に変更するとき、プロセッサはリセットされません。

電力オフ OS が、電力消費削減のため、何らかの手法を適用します。これらの手法では、プロセッサはプロセッサ状態を保持できないため、適用される手法には、消失してはならないプロセッサ状態を OS が保存する方法が含まれている必要があります。電力オフから通常動作への変更には、次の動作が含まれる必要があります。

- プロセッサのリセット。この動作は、電力レベルが回復した後で行われます。
- OS によるプロセッサ状態の復元

スタンバイは、最も侵襲性が低い、OS による電力削減状態です。この状態では、プロセッサを利用できなくなるだけで、デバッグ設定は一切クリアされません。スタンバイについて、v7 デバッグでは次の点のみが規定されています。

- プロセッサがスタンバイ状態で、ホールトデバッグ イベントがトリガされた場合、プロセッサはスタンバイを終了してデバッグイベントを処理する必要があります。プロセッサが `WFI` または `WFE` 命令を実行してスタンバイ状態に入った場合、これらの命令は終了します。
- プロセッサがスタンバイ状態で、外部デバッグまたはメモリマップされたインタフェースにアクセスが行われた場合、プロセッサは自動的に次の動作を行う必要があります。
  - — スタンバイ状態を終了する。
  - — デバッグトランザクションに対応する。
  - — スタンバイ状態に戻る。

  この動作が可能なのは、外部デバッグおよびメモリマップされたインタフェースは、PREADYDBG を LOW に保持するなどの方法により、プロセッサがスタンバイ状態を終了するまでの間ウェイト状態を挿入できるためです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The protocol for communicating between the debug logic and the power controller, enabling the processor to leave and return to standby automatically, is IMPLEMENTATION DEFINED.

v7 Debug includes features that can aid software debugging in a system that dynamically powers down the processor. These techniques are described in greater detail in the following sections.

C6.2.2 Power domains and debug

This section does not apply to v6 Debug and v6.1 Debug, which support only a single power domain.

This section discusses how, in v7 Debug, some registers can be split between different power domains to implement support for debug over power-down and re-powering of the processor.

In v7 Debug, it is IMPLEMENTATION DEFINED whether a processor supports debug over power-down:

- debug over power-down can be supported only if the processor implements the features summarized in this section
- when a processor implements the features required for debug over power-down, it is IMPLEMENTATION DEFINED whether a system that includes that processor supports debug over power-down
- usually, a system that does not support debug over power-down implements a single power domain.

An ARMv7 processor with a single power domain cannot support debug over power-down.

This means that the number of power domains supported by an ARMv7 processor is IMPLEMENTATION DEFINED. However, ARM recommends that at least two are implemented, to provide support for debug over power-down. The two power domains required for this are:

- a debug power domain
- a core power domain.

The debug power domain contains the external debug interface control logic and a subset of the debug resources. This subset is determined by physical placement constraints and other considerations that are explained later in this chapter. Figure C6-1 on page C6-7 shows an example of such a system.

For example, this arrangement is useful for debugging systems where several processors are connected to the same debug bus and where one or more of the processors can power-down at any time. It has two advantages:

- The debug bus is not made unavailable by the core power domain powering down:
  - — if the debugger tries to access the processor with the core power domain powered-down, the external debug interface can return a slave-generated error response instead of locking the system
  - — if the debugger tries to access another processor, it can proceed normally.

  The debug bus might be, for example, an APBv3 or internal debug bus.
- Some debug registers are unaffected by power-down. This means that a debugger can, for example, identify the processor while the core power domain is powered-down.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

プロセッサがスタンバイ状態の終了と復帰を自動的に行えるようにするため、デバッグロジックと電力コントローラとの間で通信を行うためのプロトコルは、実装定義です。

v7 デバッグには、プロセッサの電力を動的にオフにするシステムにおけるソフトウェアデバッグに役立つ機能が含まれています。これらの手法については、以下のセクションで詳しく説明します。

C6.2.2 電力ドメインとデバッグ

このセクションは、単一の電力ドメインのみをサポートしている v6 デバッグおよび v6.1 デバッグには適用されません。

このセクションでは、v7 デバッグで一部のレジスタを別の電力ドメインに分割し、プロセッサの電力オフと電力の回復時をまたぐデバッグのサポートを実装する方法について説明します。

v7 デバッグでは、プロセッサが電力オフをまたぐデバッグをサポートするかどうかは実装定義です。

- 電力オフをまたぐデバッグは、このセクションに概説されている機能がプロセッサに実装されている場合のみサポート可能です。
- 電力オフをまたぐデバッグに必要な機能がプロセッサに実装されている場合、そのプロセッサを含むシステムで電力オフをまたぐデバッグをサポートするかどうかは実装定義です。
- 通常は、電力オフをまたぐデバッグをサポートしないシステムでは、単一の電力ドメインが実装されます。

単一の電力ドメインを持つ ARMv7 プロセッサでは、電力オフをまたぐデバッグをサポートできません。

これは、ARMv7 プロセッサによってサポートされる電力ドメインの数は実装定義であることを意味します。ただし、電力オフをまたぐデバッグのサポートを提供するため、最低 2 つを実装することをお勧めします。このために必要なのは、次の 2 つの電力ドメインです。

- デバッグ電力ドメイン
- コア電力ドメイン

デバッグ電力ドメインには、外部デバッグインタフェースの制御ロジックと、デバッグリソースのサブセットが含まれます。このサブセットは、本章の以降で説明する、物理的な配置の制限および他の考慮事項により決定されます。このようなシステムの例を、P. C6-7 の図 C6-1 に示します。

例えば、この配置は、複数のプロセッサが同じデバッグバスに接続され、1 つ以上のプロセッサがいつでも電力オフ可能なシステムのデバッグに適しています。この配置には 2 つの利点があります。

- デバッグバスが、コア電力ドメインの電力オフによって利用できなくなることがありません。
  - – コア電力ドメインの電力がオフになっているプロセッサへ、デバッガがアクセスを試みると、外部デバッグインタフェースはシステムをロックする代わりに、スレーブで生成されるエラー応答を返すことができます。
  - – デバッガが他のプロセッサへアクセスを試みると、通常に動作が行われます。

  デバッグバスに、例えば APBv3 や内部デバッグバスを使用できます。
- 一部のデバッグレジスタは、電力オフの影響を受けません。つまり、デバッガは、コア電力ドメインの電力がオフになっているプロセッサを識別するなどの動作を実行できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

To have full debug support for power-down and re-powering of the processor, the following registers and individual bits need to be in the debug power domain:

DBGECR This enables the debugger to set the OS Unlock Catch bit to 1 any time and still break on completion of the power-up sequence. If this register was in the core power domain, the power-down event would clear this catch bit to 0. For more information, see *Event Catch Register (DBGECR)* on page C10-78.

DBGDRCR[0] Halt request bit

This enables the debugger to request a Debug state entry even if the processor is powered down. Also, if the debugger makes this request before powering-down but the request cannot be satisfied, for example because the processor is in Secure state but (**DBGEN** AND **SPIDEN**) = 0, the request remains pending through power-down.

Note

The processor has to be powered up to respond to a pending DBGDRCR[0] Halt request or External Debug request.

OS Lock Access Register

This enables the lock that the OS sets before saving the debug registers to remain set through power-down. For details see *OS Lock Access Register (DBGOSLAR)* on page C10-75.

Device Power-down and Reset registers

These registers must be in the debug power domain because some of their functions are used for debugging power-down events. See *Device Power-down and Reset Control Register (DBGPRCR), v7 Debug only* on page C10-31, *Device Power-down and Reset Status Register (DBGPRSR), v7 Debug only* on page C10-34.

Lock Access Register, if implemented

If implemented, this register must be in the debug power domain because it is used to enable certain accesses by external debug interface, and this functionality is required when debugging power-down events.

Identification registers and the DBGDIDR

The identification registers are at addresses `0xD00-0xDFC`, and `0xFD0-0xFEC`. For details of these registers see *Management registers, ARMv7 only* on page C10-88.

Debugger operation only requires the above registers and bits to be in the debug power domain. However, to rationalize the split between the debug and core power domains in the register map, ARMv7 requires an implementation that supports debug over power-down to have all bits of the following registers in the debug power domain:

DBGDIDR, DBGECR, and DBGDRCR

No error response is returned on read or write accesses when the core power domain is powered down.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

プロセッサの電力オフと電力の回復について完全にデバッグをサポートするには、デバッグ電力ドメインに次のレジスタと各ビットが必要です。

DBGECR 電力オンシーケンスの完了時のたびに、デバッガが OS ロック解除キャッチビットを 1 にセットし、保持するために使用されます。このレジスタがコア電力ドメインにある場合、電力オフイベントによってこのキャッチビットは 0 にクリアされます。詳細については、*イベントキャッチ レジスタ (DBGECR)* P. C10-78 を参照して下さい。

DBGDRCR[0]、ホールト要求ビット

プロセッサの電力がオフのときでも、デバッガがデバッグ状態の開始を要求できるようにするために使用されます。また、電力オフ前にデバッガがこの要求を発行したが、その要求を実行できなかった場合、例えばプロセッサがセキュア状態であるが (DBGEN AND SPIDEN) = 0 の場合、電力オフの間も要求は保留中に保たれます。

注

プロセッサが保留中の DBGDRCR[0] ホールト要求や外部デバッグ要求に応答するには、電力がオンになる必要があります。

OS ロックアクセス レジスタ

デバッグレジスタの保存前に OS がセットしたロックを、電力オフの間もセット状態に維持するため使用されます。詳細については、*OS ロックアクセス レジスタ (DBGOSLAR)* P. C10-75 を参照して下さい。

デバイス電力オフおよびリセットレジスタ

これらのレジスタは、いくつかの機能が電力オフイベントのデバッグに使用されるため、デバッグ電力ドメインに存在する必要があります。詳細については、*デバイス電力オフおよびリセット制御レジスタ (DBGPRCR)、v7 デバッグのみ* P. C10-31 および*デバイス電力オフおよびリセット ステータス レジスタ (DBGPRSR)、v7 デバッグのみ* P. C10-34 を参照して下さい。

ロックアクセス レジスタ（実装されている場合）

このレジスタが実装されている場合、デバッグ電力ドメインに存在する必要があります。このレジスタは、外部デバッグインタフェースによる特定のアクセスを可能にするために使用され、電力オフイベントをデバッグするために必要となるためです。

識別レジスタと DBGDIDR

識別レジスタは、アドレス `0xD00` ～ `0xDFC` と `0xFD0` ～ `0xFEC` にあります。これらのレジスタの詳細については、*管理レジスタ (ARMv7 のみ)* P. C10-88 を参照して下さい。

デバッガの動作に必要なのは、これらのレジスタとビットがデバッグ電力ドメインに存在することのみです。ただし、レジスタマップでデバッグおよびコア電力ドメインを論理的に分割するため、ARMv7 では、電力オフをまたぐデバッグをサポートする実装について、次に示すレジスタのすべてのビットがデバッグ電力ドメインに存在することが要求されます。

DBGDIDR、DBGECR、DBGDRCR

コア電力ドメインの電力がオフになっているとき、読み出しまたは書き込みアクセスでエラー応答は返されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

OS Save and Restore registers, and Device Power-down and Reset registers

No error response returned on read or write accesses when the core power domain is powered down. However, accesses to the OS Lock Access Register (DBGOSLAR) and OS Save and Restore Register (DBGOSSRR) are UNPREDICTABLE when the core power domain is powered-down.

All of the management registers, except for the IMPLEMENTATION DEFINED integration registers

The management registers are registers 823 - 1023, in the address range `0xD00-0xFFC`. Requiring all these registers to be in the debug power domain simplifies the decoding of register addresses for the registers in the debug power domain.

Note

The CP15 c0 registers (`0xD00-0xDFC`) are included in this category.

For all other registers, including any IMPLEMENTATION DEFINED registers, it is IMPLEMENTATION DEFINED whether the register is implemented in the core or the debug power domain.

Figure C6-1 shows the recommended power domain split.

Figure C6-1 Recommended power domain split between core and debug power domains

The signals **DBGNOPWRDWN** and **DBGPWRDUP** shown in Figure C6-1 above form an interface between the power controller and the processor debug logic that is in the debug power domain. With this interface:

- the external debugger can request the power controller to emulate power-down, simplifying the requirements on software by sacrificing entirely realistic behavior

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

OS 保存および復元レジスタと、デバイス電力オフおよびリセットレジスタ

コア電力ドメインの電力がオフになっているとき、読み出しまたは書き込みアクセスでエラー応答は返されません。ただし、OS ロックアクセス レジスタ (DBGOSLAR) と、OS 保存および復元レジスタ (DBGOSSRR) へのアクセスは、コア電力ドメインの電力がオフのときは予測不能です。

実装定義の統合レジスタを除く、すべての管理レジスタ

管理レジスタはレジスタ 823 ～ 1023 で、アドレス範囲 `0xD00` ～ `0xFFC` にあります。これらすべてのレジスタがデバッグ電力ドメインにあることを必須にすると、デバッグ電力ドメインのレジスタについてレジスタアドレスのデコードが簡単になります。

注

CP15 c0 レジスタ (`0xD00` ～ `0xDFC`) はこのカテゴリに含まれています。

実装定義のレジスタも含む、他のすべてのレジスタについて、コア電力ドメインとデバッグ電力ドメインのどちらに実装されるかは実装定義です。

推奨される電力ドメインの分割を、図 C6-1 に示します。

図 C6-1 推奨されるコアおよびデバッグ電力ドメインの分割

上の図 C6-1 に示されている信号 DBGNOPWRDWN および DBGPWRDUP は、電力コントローラと、デバッグ電力ドメインにあるプロセッサのデバッグロジックとのインタフェースを構成します。このインタフェースにより、次の動作が可能になります。

- 外部デバッガは、電力オフをエミュレートするよう電力コントローラに要求でき、完全に現実的な動作ではない替わりにソフトウェアの要件が簡素化されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- the external debug interface knows when the core power domain is powered down, and can communicate this information to the external debugger.

For details of these signals see *DBGNOPWRDWN* on page AppxA-9 and *DBGPWRDUP* on page AppxA-10.

If the core power domain is not being powered down at the same time as the debug power domain then the authentication signal **DBGEN** must be pulled LOW before power is removed from the debug power domain. The behavior of the debug logic, and in particular the generation of debug events, is UNPREDICTABLE when the debug power domain is not powered if **DBGEN** is not LOW. Pulling **DBGEN** LOW ensures that debug events are ignored by the processor. For more information, see *Changing the authentication signals* on page AppxA-4.

Reads and writes of debug registers when the debug logic is powered down are UNPREDICTABLE.

The performance monitors must be implemented in the core power domain, and must continue to operate when debug power is removed.

The rest of this part of this manual assumes that two power domains are implemented as described in this section, and that therefore the implementation supports debug over power-down. Features that are not required for an ARMv7 implementation with a single power domain are identified as *SinglePower*, with a description of the differences in behavior. A SinglePower implementation cannot support debug over power-down.

C6.2.3 The OS Save and Restore mechanism

The OS Save and Restore mechanism enables an operating system to save the debug registers before power-down and restore them when power is restored. This extends the support for debug over power-down, and permits debug tools to work at a higher level of abstraction when there are no power-down events.

In v7 Debug:

- If an implementation supports debug over power-down, then it must implement the OS Save and Restore mechanism.
- On a SinglePower implementation, and on any other implementation that does not support debug over power-down, it is IMPLEMENTATION DEFINED whether the OS Save and Restore mechanism is implemented.
- If the OS Save and Restore mechanism is not implemented, the DBGOSLSR must be implemented as RAZ, and the other OS Save and Restore mechanism register encodings must be RAZ/WI.

In v6 Debug and v6.1 Debug, these registers are not defined.

Two of the requirements for an implementation that supports debug over power-down are:

- An operating system must be able to save and restore the much of the debug logic state over a power-down. This requirement is met by the OS Save and Restore mechanism.
- A debugger must be able to detect that a processor has powered-down. For more information, see *Permissions in relation to power-down* on page C6-28.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- 外部デバッグインタフェースは、コア電力ドメインの電力がオフになっていることを認識でき、この情報を外部デバッグに伝達できます。

これらの信号の詳細については、*DBGNOPWRDWN* P. 付録 A-9 および *DBGPWRDUP* P. 付録 A-10 を参照して下さい。

コア電力ドメインが、デバッグ電力ドメインと同時に電力オフされない場合、デバッグ電力ドメインの電力がオフになる前に、認証信号 DBGEN を LOW にプルする必要があります。DBGEN が LOW でない場合、デバッグ電力ドメインの電力がオフになっているときのデバッグロジックの動作、特にデバッグイベントの生成は予測不能です。DBGEN を LOW にプルすると、プロセッサによりデバッグイベントが無視されることが保証されます。詳細については、*認証信号の変更* P. 付録 A-4 を参照して下さい。

デバッグロジックの電力がオフのとき、デバッグレジスタへの読み出しおよび書き込みは予測不能です。

パフォーマンスモニタはコア電力ドメインに実装され、デバッグ電力がオフの間は動作し続ける必要があります。

本書の以後の部分では、2 つの電力ドメインが、このセクションで説明したように実装され、それによって電力オフをまたぐデバッグが実装でサポートされていることを想定します。単一の電力ドメインを持つ ARMv7 実装では必要ない機能は *SinglePower* として示され、動作の相違点について説明されます。SinglePower 実装では、電力オフをまたぐデバッグをサポートできません。

C6.2.3 OS 保存および復元機構

OS 保存および復元機構により、オペレーティングシステムは電力オフの前にデバッグレジスタを保存し、電力の復元後にそれらのレジスタを復元できます。これによって、電力オフをまたぐデバッグのサポートが拡張され、電力オフイベントがない場合はデバッグツールがより高い抽象化レベルで動作することが可能になります。

v7 デバッグでは、次の条件が適用されます。

- 実装で電力オフをまたぐデバッグをサポートする場合、その実装には OS 保存および復元機構を実装する必要があります。
- SinglePower 実装、および電力オフをまたぐデバッグをサポートしない他の実装では、OS 保存および復元機構を実装するかどうかは実装定義です。
- OS 保存および復元機構が実装されていない場合、DBGOSLSR は RAZ として実装する必要があり、他の OS 保存および復元機構レジスタのエンコードは RAZ/WI の必要があります。

v6 デバッグおよび v6.1 デバッグでは、これらのレジスタは定義されていません。

電力オフをまたぐデバッグをサポートする実装には、次の 2 つの必要条件があります。

- オペレーティングシステムは、電力オフの前後で、デバッグロジックの状態の多くを保存および復元することが可能である必要があります。この条件は、OS 保存および復元機構によって満たされます。
- デバッガは、プロセッサの電力がオフになっていることを検出可能な必要があります。詳細については、*電力オフとアクセス許可との関係* P. C6-27 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The OS Save and Restore mechanism is provided by the following registers:

- OS Save and Restore Register (DBGOSSRR), see *OS Save and Restore Register (DBGOSSRR)* on page C10-77
- OS Lock Access Register (DBGOSLAR), see *OS Lock Access Register (DBGOSLAR)* on page C10-75
- OS Lock Status Register (DBGOSLSR), see *OS Lock Status Register (DBGOSLSR)* on page C10-76
- The Event Catch Register (DBGECR), see *Event Catch Register (DBGECR)* on page C10-78.

You can read the DBGOSLSR to detect whether the OS Save and Restore mechanism is implemented. If it is not implemented the read of the DBGOSLSR returns zero.

The DBGOSSRR works in conjunction with an internal sequence counter, so that a series of reads or writes of this register saves or restores the complete debug logic state of the processor that would be lost when the processor is powered down. The internal sequence counter is reset to the start of the sequence by writing the key, `0xC5ACCE55`, to the DBGOSLAR.

The number of accesses required, and the order and interpretation of the data are IMPLEMENTATION DEFINED.

The first access to the DBGOSSRR following the reset of the internal sequence counter must be a read:

- when performing an OS Save sequence this read returns the number of reads from the DBGOSSRR that are needed to save the entire debug logic state
- when performing an OS Restore sequence the value returned by this read is UNKNOWN.

The result of issuing a write to the DBGOSSRR following a reset of the internal sequence counter is UNPREDICTABLE.

Note

- If the OS Save and Restore mechanism is not implemented, this first read returns zero, correctly indicating to software that no registers are to be saved.
- An implementation that includes the OS Save and Restore mechanism might not provide access to the DBGOSSRR through the external debug interface. In this case:
  - the DBGOSLSR, DBGOSLAR, and DBGECR are accessible through the external debug interface
  - through the external debug interface, the DBGOSSRR is RAZ/WI
  - because the first read of the DBGOSSRR through the external debug interface returns zero, this indicates that the OS registers cannot be saved or restored through the external debug interface.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

OS 保存および復元機構は、次に示すレジスタによって提供されます。

- OS 保存および復元レジスタ (DBGOSSRR)。*OS 保存および復元レジスタ (DBGOSSRR)* P. C10-77 を参照して下さい。
- OS ロックアクセス レジスタ (DBGOSLAR)。*OS ロックアクセス レジスタ (DBGOSLAR)* P. C10-75 を参照して下さい。
- OS ロックステータス レジスタ (DBGOSLSR)。*OS ロックステータス レジスタ (DBGOSLSR)* P. C10-76 を参照して下さい。
- イベントキャッチ レジスタ (DBGECR)。*イベントキャッチ レジスタ (DBGECR)* P. C10-78 を参照して下さい。

DBGOSLSR を読み出すと、OS 保存および復元機構が実装されているかどうかを検出できます。実装されていない場合、DBGOSLSR を読み出すと 0 が返されます。

DBGOSSRR は、内部シーケンスカウンタとの組み合わせで動作するため、このレジスタへの一連の読み出しまたは書き込みにより、プロセッサの電力がオフになったときに消失する、完全なデバッグロジック状態を保存または復元できます。内部シーケンスカウンタは、DBGOSLAR にキー `0xC5ACCE55` を書き込むことで、シーケンスの開始にリセットされます。

必要なアクセスの回数と、データの解釈の順序は実装定義です。

内部シーケンスカウンタのリセット後、DBGOSSRR への最初のアクセスでは、次の内容が読み出される必要があります。

- OS 保存シーケンスを実行しているとき、この読み出しでは、デバッグロジック状態の全体を保存するために必要な、DBGOSSRR からの読み出し回数が返されます。
- OS 復元シーケンスを実行しているとき、この読み出しで返される値は未知です。

内部シーケンスカウンタのリセット後に DBGOSSRR へ書き込みを発行した結果は予測不能です。

注

- OS 保存および復元機構が実装されていない場合、最初の読み出しで 0 が返され、保存すべきレジスタはないことがソフトウェアに対して正しく示されます。
- OS 保存および復元機構を含む実装で、外部デバッグインタフェースを経由した DBGOSSRR へのアクセスが提供されていないことがあります。この場合、次の動作が行われます。
  - DBGOSLSR、DBGOSLAR、DBGECR は、外部デバッグインタフェース経由でアクセス可能です。
  - 外部デバッグインタフェース経由では、DBGOSSRR は RAZ/WI です。
  - 外部デバッグインタフェース経由の DBGOSSRR への最初の読み出しは 0 を返すため、外部デバッグインタフェース経由で OS レジスタの保存または復元ができないことが示されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The subsequent accesses to the DBGOSSRR must be either all reads or all writes. UNPREDICTABLE behavior results if:

- reads and writes are mixed
- more accesses are performed than the number of registers to be saved or restored, as returned by the first read in the OS Save sequence.
- the subsequent accesses are writes, but the OS Lock is cleared with fewer writes performed than the number of registers to be restored.

The debug logic state of the processor is unchanged if the OS Lock is cleared during or following an OS Save sequence. The sequence is restarted the next time the OS Lock is set.

When the core power domain is powered down or when the OS Lock is not locked, reads of DBGOSSRR return an UNKNOWN value and writes are UNPREDICTABLE.

See *Example OS Save and Restore sequences* on page C6-12 for software examples of the OS Save and Restore processes.

The debug logic state preserved by the OS Save and Restore mechanism

If debug over power-down is supported, the OS Save and Restore mechanism permits the following debug logic state to be preserved:

- The registers that must be in the debug power domain, see *Power domains and debug* on page C6-5.
- The DBGWFAR.
- The DBGBVRs, DBGBCRs, DBGWVRs, DBGWCRs, and DBGVCR.
- The DBGDSCCR and DBGDSMCR.
- The data transfer registers DBGDTRTX and DBGDTRRX, subject to the values of DBGDSCR.TXfull and DBGDSCR.RXfull when the OS Save sequence is performed:
  - If DBGDSCR.TXfull is set to 1 then the value of DBGDTRTX is guaranteed to be saved and restored.
  - If DBGDSCR.RXfull is set to 1 then the value of DBGDTRRX is guaranteed to be saved and restored.
  - If either of these flags is not set to 1 when the OS Save sequence is performed then the value of the corresponding register is UNKNOWN after the OS Restore sequence.

Note

The OS Save and Restore sequences must not stall reading the values of DBGDTRTX and DBGDTRRX, and must not cause any instructions to be issued, regardless of the settings of the DBGDSCR.ExtDCCmode access mode bits.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

DBGOSSRR への以後のアクセスは、すべて読み出しか、すべて書き込みの必要があります。次の場合、動作は予測不能です。

- 読み出しと書き込みが混在している。
- OS 保存シーケンスで最初の読み出しにより返される、保存または復元すべきレジスタの数よりも多いアクセスが行われた。
- 以後のアクセスが書き込みだが、復元すべきレジスタの数よりも少ない回数の書き込みを実行後に OS ロックがクリアされた。

OS 保存シーケンスの間、またはその後で OS ロックがクリアされた場合、プロセッサのデバッグロジック状態は変化しません。このシーケンスは、次に OS ロックが設定されたときに再開されます。

コア電力ドメインの電力がオフになったとき、または OS ロックがロックされていないとき、DBGOSSRR の読み出しにより返される値は未知で、書き込みは予測不能です。

OS 保存および復元プロセスのソフトウェアの例については、*OS 保存および復元シーケンスの例* P. C6-12 を参照して下さい。

OS 保存および復元機構により保持されるデバッグロジック状態

電力オフをまたぐデバッグがサポートされている場合、OS 保存および復元機構では、デバッグロジックの次の状態を保持することができます。

- デバッグ電力ドメインに存在することが必要なレジスタ。*電力ドメインとデバッグ* P. C6-5 を参照して下さい。
- DBGWFAR
- DBGBVR、DBGBCR、DBGWVR、DBGWCR、DBGVCR
- DBGDSCCR と DBGDSMCR
- データ転送レジスタ DBGDTRTX および DBGDTRRX。これらは、OS 保存シーケンスが実行されるときの DBGDSCR.TXfull および DBGDSCR.RXfull の値に依存します。
  - DBGDSCR.TXfull が 1 にセットされている場合、DBGDTRTX の値が必ず保存および復元されます。
  - DBGDSCR.RXfull が 1 にセットされている場合、DBGDTRRX の値が必ず保存および復元されます。
  - OS 保存シーケンスの実行時にこれらのフラグのいずれかが 1 にセットされていない場合、OS 復元シーケンスの後で、対応するレジスタの値は未知です。

注

OS 保存および復元シーケンスは、DBGDTRTX および DBGDTRRX の値を読み出すときにストールできません。また、DBGDSCR.ExtDCCmode アクセスモード ビットの設定にかかわらず、一切の命令発行を引き起こすことはできません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- The DCC status flags themselves:
  - DBGDSCR.TXfull, bit [29]
  - DBGDSCR.TXfull_l, bit [26]
  - DBGDSCR.RXfull, bit [30]
  - DBGDSCR.RXfull_l, bit [27].

  Note

  Reading DBGDSCR through the DBGOSSRR has no side-effects, that is, the values of TXfull_l and RXfull_l are unchanged.

- All other writable flags in the DBGDSCR:
  - Method of Debug Entry bits, MOE, bits [5:2]
  - Force Debug Acknowledge bit, DBGack, bit [10]
  - Interrupts Disable bit, INTdis, bit [11]
  - User mode Access to Communication Channel Enable bit, UDCCdis, bit [12]
  - Execute ARM Instruction Enable bit, ITRen, bit [13]
  - Halting debug-mode Enable bit, HDBGen, bit [14]
  - Monitor debug-mode Enable bit, MDBGen, bit [15]
  - External DCC access mode field, ExtDCCmode, bits [21:20].

- If vectored interrupt support is implemented and enabled, all state required to ensure the correct generation of Vector Catch debug events. For more information, see *Vector catch debug events and vectored interrupt support* on page C3-22.

The OS Save sequence must preserve at least all of this debug logic state that is lost when the core power domain is powered down. The OS Save sequence does not have to preserve any debug logic state that is not lost when the core power domain is powered down. That is, it does not have to preserve any debug logic state that is in the debug power domain.

The OS Save and Restore mechanism does not preserve:

- The sticky exception flags in the DBGDSCR, and the contents of the DBGITR.
- The read-only processor status flags in the DBGDSCR:
  - HALTED, bit [0]
  - RESTARTED, bit [1]
  - SPIDdis, bit [16]
  - SPNIDdis, bit [17]
  - NS, bit [18]
  - ADAdiscard, bit [19]
  - InstrCompl_l, bit [24]
  - PipeAdv, bit [25].

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- DCC ステータスフラグ自体
  - DBGDSCR.TXfull、ビット [29]
  - DBGDSCR.TXfull_l、ビット [26]
  - DBGDSCR.RXfull、ビット [30]
  - DBGDSCR.RXfull_l、ビット [27]

  注

  DBGOSSRR 経由で DBGDSCR を読み出しても副作用はありません。つまり、TXfull_l および RXfull_l の値は変更されません。

- DBGDSCR に含まれている他のすべての書き込み可能フラグ
  - デバッグ開始方法ビット (MOE)、ビット [5:2]
  - 強制デバッグ応答ビット (DBGack)、ビット [10]
  - 割り込みディセーブルビット (INTdis)、ビット [11]
  - 通信チャネルへのユーザモードアクセス イネーブルビット (UDCCdis)、ビット [12]
  - ARM 命令実行イネーブルビット (ITRen)、ビット [13]
  - ホールトデバッグモード イネーブルビット (HDBGen)、ビット [14]
  - モニタデバッグモード イネーブルビット (MDBGen)、ビット [15]
  - 外部 DCC アクセスモード フィールド (ExtDCCmode)、ビット [21:20]

- ベクタ割り込みサポートが実装され、可能な場合、ベクタキャッチ デバッグイベントの正しい生成を保証するため必要なすべての状態。詳細については、*ベクタキャッチデバッグ イベントとベクタ割り込みのサポート* P. C3-22 を参照して下さい。

OS 保存シーケンスは、最低でも、コア電力ドメインの電力がオフになったとき消失する、これらのデバッグロジックの状態をすべて保持する必要があります。OS 保存シーケンスは、コア電力ドメインの電力がオフになったときに消失しない、デバッグロジックの状態を保持する必要はありません。つまり、デバッグ電力ドメインのデバッグロジック状態を保持する必要はありません。

OS 保存および復元機構は、次の状態を保持しません。

- DBGDSCR のスティッキー例外フラグ、および DBGITR の内容
- DBGDSCR の読み出し専用プロセッサステータス フラグ
  - HALTED、ビット [0]
  - RESTARTED、ビット [1]
  - SPIDdis、ビット [16]
  - SPNIDdis、ビット [17]
  - NS、ビット [18]
  - ADAdiscard、ビット [19]
  - InstrCompl_l、ビット [24]
  - PipeAdv、ビット [25]

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- The performance monitor registers described in Chapter C9 *Performance Monitors*.
- The trace registers.

The OS Restore sequence always overwrites the debug registers with the values that were saved. In particular, the values of the DBGDTRTX and DBGDTRRX Registers, and of the DCC status flags TXfull, TXfull_l, RXfull, and RXfull_l after the OS Restore sequence are the saved values.

If there were valid values in the DBGDTRTX or DBGDTRRX Registers immediately before the OS Restore sequence then those values are lost.

Example OS Save and Restore sequences

Example OS Save and Restore sequences are described in:

- *Example OS Save and Restore sequences using the memory-mapped interface*
- *Example OS Save and Restore sequences using the Extended CP14 interface* on page C6-14.

Example OS Save and Restore sequences using the memory-mapped interface

On an implementation that includes the OS Save and Restore mechanism and a memory-mapped interface:

- Example C6-1 shows the correct sequence for saving the debug logic state, using the memory-mapped interface, before powering down
- Example C6-2 on page C6-13 shows the correct sequence for restoring the debug logic state, using the memory-mapped interface, when the system is powered on again.

When the debug logic state is restored, if the OS Unlock Catch bit in the Event Catch Register is set to 1 a debug event is triggered when the DBGOSLAR is cleared. This event might be used by an external debugger to restart a debugging session. See *Event Catch Register (DBGECR)* on page C10-78.

Example C6-1 OS debug register save sequence, memory-mapped interface

```
; On entry, R0 points to a block to save the debug registers in.

SaveDebugRegisters
        PUSH    {R4, LR}
        MOV     R4, R0                          ; Save pointer

        ; (1) Set OS Lock Access Register (DBGOSLAR). The architecture requires that DBGOSLAR
        ;     and the other debug registers have at least the Device memory attribute.
        BL      GetDebugRegisterBase            ; Returns base in R0
        LDR     R1, =0xC5ACCE55
        STR     R1, [R0, #0x300]                ; Write DBGOSLAR

        ; (2) Get the number of words to save.
        LDR     R1, [R0, #0x308]                ; DBGOSSRR returns size
        STR     R1, [R4], #4                    ; Push on to the save stack

        ; (3) Loop reading words from the DBGOSSRR.
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- 第 C9 章「*パフォーマンスモニタ*」で説明されているパフォーマンスモニタ レジスタ
- トレースレジスタ

OS 復元シーケンスは、デバッグレジスタを、保存されていた値で常に上書きします。特に、DBGDTRTX および DBGDTRRX レジスタの値、および DCC ステータスフラグ TXfull、TXfull_l、RXfull、RXfull_l の値は、OS 復元シーケンスの後では保存されていた値になります。

OS 復元シーケンスの直前に DBGDTRTX または DBGDTRRX レジスタに有効な値が格納されていた場合、それらの値は消失します。

OS 保存および復元シーケンスの例

OS 保存および復元シーケンスの例については、次に示すセクションを参照して下さい。

- *メモリマップされたインタフェースを使用する OS 保存および復元シーケンスの例*
- *拡張 CP14 インタフェースを使用する OS 保存および復元シーケンスの例* P. C6- 14

メモリマップされたインタフェースを使用する OS 保存および復元シーケンスの例

OS 保存および復元機構と、メモリマップされたインタフェースを含む実装については、次に示す例を参照して下さい。

- 電力オフの前にメモリマップされたインタフェースを使用してデバッグロジックの状態を保存する正しいシーケンスを、例 C6- 1 に示します。
- システムの電力が再度オンになった後で、メモリマップされたインタフェースを使用してデバッグロジックの状態を復元する正しいシーケンスを、P. C6- 13 の例 C6- 2 に示します。

デバッグロジックの状態を復元するとき、イベントキャッチ レジスタの OS ロック解除キャッチビットが 1 にセットされていると、DBGOSLAR がクリアされたときにデバッグイベントがトリガされます。このイベントは、外部デバッガがデバッグセッションを再開するために使用できます。*イベントキャッチ レジスタ (DBGECR)* P. C10- 78 を参照して下さい。

例 C6- 1 OS デバッグレジスタの保存シーケンス、メモリマップされたインタフェース

```
; On entry, R0 points to a block to save the debug registers in.

SaveDebugRegisters
        PUSH    {R4, LR}
        MOV     R4, R0                          ; Save pointer

        ; (1) Set OS Lock Access Register (DBGOSLAR). The architecture requires that DBGOSLAR
        ;     and the other debug registers have at least the Device memory attribute.
        BL      GetDebugRegisterBase            ; Returns base in R0
        LDR     R1, =0xC5ACCE55
        STR     R1, [R0, #0x300]                ; Write DBGOSLAR

        ; (2) Get the number of words to save.
        LDR     R1, [R0, #0x308]                ; DBGOSSRR returns size
        STR     R1, [R4], #4                    ; Push on to the save stack

        ; (3) Loop reading words from the DBGOSSRR.
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        CMP     R1, #0                          ; Check for zero
SaveDebugRegisters_Loop
        ITTT    NE
        LDRNE   R2, [R0, #0x308]                ; Load a word of data
        STRNE   R2, [R4], #4                    ; Push on to the save stack
        SUBSNE  R1, R1, #1
        BNE     SaveDebugRegisters_Loop

        ; (4) Return the pointer to first word not written to. Leave DBGOSLAR set, because
        ;     from now on we do not want any changes.
        MOV     R0, R4
        POP     {R4, PC}
```

Example C6-2 OS debug register restore sequence, memory-mapped interface

```
; On entry, R0 points to a block of saved debug registers.

RestoreDebugRegisters
        PUSH    {R4, LR}
        MOV     R4, R0                          ; Save pointer

        ; (1) Set the OS Lock Access Register (DBGOSLAR) and reset pointer. The lock
        ;     will already be set, but this write is needed to reset the pointer. The
        ;     architecture requires that DBGOSLAR and the other debug registers have at
        ;     least the Device memory attribute.
        BL      GetDebugRegisterBase            ; Returns base in R0
        LDR     R1, =0xC5ACCE55
        STR     R1, [R0, #0x300]                ; Write DBGOSLAR

        ; (2) Clear the Sticky Power-down Status bit.
        LDR     R1, [R0, #0x314]                ; Read DBGPRSR to clear StickyPD

        ; (3) Get the number of words saved.
        LDR     R1, [R0, #0x308]                ; Dummy read of DBGOSSRR
        LDR     R1, [R4], #4                    ; Get register count from the save stack

        ; (4) Loop writing words from the DBGOSSRR.
        CMP     R1, #0                          ; Check for zero
RestoreDebugRegisters_Loop
        ITTT    NE
        LDRNE   R2, [R4], #4                    ; Load a word from the save stack
        STRNE   R2, [R0, #0x308]                ; Store a word of data
        SUBSNE  R1, R1, #1
        BNE     RestoreDebugRegisters_Loop

        ; (5) Clear the DBGOSLAR. Writing any non-key value clears the lock, so use the
        ;     zero value in R1.
        STR     R1, [R0, #0x300]                ; Write DBGOSLAR

        ; (6) A final DSB ensures the restore is complete and an ISB ensures
        ;     the restored register values are visible to subsequent instructions.
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
        CMP     R1, #0                          ; Check for zero
SaveDebugRegisters_Loop
        ITTT    NE
        LDRNE   R2, [R0, #0x308]                ; Load a word of data
        STRNE   R2, [R4], #4                    ; Push on to the save stack
        SUBSNE  R1, R1, #1
        BNE     SaveDebugRegisters_Loop

        ; (4) Return the pointer to first word not written to. Leave DBGOSLAR set, because
        ;     from now on we do not want any changes.
        MOV     R0, R4
        POP     {R4, PC}
```


例 C6-2 OS デバッグレジスタの復元シーケンス、メモリマップされたインタフェース


```
; On entry, R0 points to a block of saved debug registers.

RestoreDebugRegisters
        PUSH    {R4, LR}
        MOV     R4, R0                          ; Save pointer

        ; (1) Set the OS Lock Access Register (DBGOSLAR) and reset pointer. The lock
        ;     will already be set, but this write is needed to reset the pointer. The
        ;     architecture requires that DBGOSLAR and the other debug registers have at
        ;     least the Device memory attribute.
        BL      GetDebugRegisterBase            ; Returns base in R0
        LDR     R1, =0xC5ACCE55
        STR     R1, [R0, #0x300]                ; Write DBGOSLAR

        ; (2) Clear the Sticky Power-down Status bit.
        LDR     R1, [R0, #0x314]                ; Read DBGPRSR to clear StickyPD

        ; (3) Get the number of words saved.
        LDR     R1, [R0, #0x308]                ; Dummy read of DBGOSSRR
        LDR     R1, [R4], #4                    ; Get register count from the save stack

        ; (4) Loop writing words from the DBGOSSRR.
        CMP     R1, #0                          ; Check for zero
RestoreDebugRegisters_Loop
        ITTT    NE
        LDRNE   R2, [R4], #4                    ; Load a word from the save stack
        STRNE   R2, [R0, #0x308]                ; Store a word of data
        SUBSNE  R1, R1, #1
        BNE     RestoreDebugRegisters_Loop

        ; (5) Clear the DBGOSLAR. Writing any non-key value clears the lock, so use the
        ;     zero value in R1.
        STR     R1, [R0, #0x300]                ; Write DBGOSLAR

        ; (6) A final DSB ensures the restore is complete and an ISB ensures
        ;     the restored register values are visible to subsequent instructions.
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
        DSB
        ISB

        ; (7) Return the pointer to first word not read.
        MOV     R0, R4
        POP     {R4, PC}
```

Example OS Save and Restore sequences using the Extended CP14 interface

On an implementation that includes the OS Save and Restore mechanism and the Extended CP14 interface:

- Example C6-3 shows the correct sequence for saving the debug logic state, using the Extended CP14 interface, before powering down
- Example C6-4 on page C6-15 shows the correct sequence, using the Extended CP14 interface, for restoring the debug logic state when the system is powered on again.

When the debug logic state is restored, if the OS Unlock Catch bit in the Event Catch Register is set to 1 a debug event is triggered when the DBGOSLAR is cleared. This event might be used by an external debugger to restart a debugging session. See *Event Catch Register (DBGECR)* on page C10-78.

Example C6-3 OS debug register save sequence, Extended CP14 interface

```
; On entry, R0 points to a block to save the debug registers in.

SaveDebugRegisters
        ; (1) Set OS Lock Access Register (DBGOSLAR).
        LDR     R1, =0xC5ACCE55
        MCR     p14, 0, R1, c1, c0, 4           ; Write DBGOSLAR
        ISB

        ; (2) Get the number of words to save.
        MRC     p14, 0, R1, c1, c2, 4           ; DBGOSSRR returns size
        STR     R1, [R0], #4                    ; Push on to the save stack

        ; (3) Loop reading words from the DBGOSSRR.
        CMP     R1, #0                          ; Check for zero
SaveDebugRegisters_Loop
        ITTT    NE
        MRCNE   p14, 0, R2, c1, c2, 4           ; Load a word of data
        STRNE   R2, [R0], #4                    ; Push on to the save stack
        SUBSNE  R1, R1, #1
        BNE     SaveDebugRegisters_Loop

        ; (4) Return the pointer to first word not written to. This pointer is already in R0, so
        ;     all that is needed is to return from this function.
        ;
        ;     Leave DBGOSLAR set, because from now on we do not want any changes.
        BX      LR
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳


```
        DSB
        ISB

        ; (7) Return the pointer to first word not read.
        MOV     R0, R4
        POP     {R4, PC}
```


拡張 CP14 インタフェースを使用する OS 保存および復元シーケンスの例

OS 保存および復元機構と、拡張 CP14 インタフェースを含む実装については、次に示す例を参照して下さい。

- 電力オフの前に拡張 CP14 インタフェースを使用してデバッグロジックの状態を保存する正しいシーケンスを、例 C6-3 に示します。
- システムの電力が再度オンになった後で、拡張 CP14 インタフェースを使用してデバッグロジックの状態を復元する正しいシーケンスを、P. C6-15 の例 C6-4 に示します。

デバッグロジックの状態を復元するとき、イベントキャッチ レジスタの OS ロック解除キャッチビットが 1 にセットされていると、DBGOSLAR がクリアされたときにデバッグイベントがトリガされます。このイベントは、外部デバッガがデバッグセッションを再開するために使用できます。*イベントキャッチ レジスタ (DBGECR)* P. C10-78 を参照して下さい。

例 C6-3 OS デバッグレジスタの保存シーケンス、拡張 CP14 インタフェース


```
; On entry, R0 points to a block to save the debug registers in.

SaveDebugRegisters
        ; (1) Set OS Lock Access Register (DBGOSLAR).
        LDR     R1, =0xC5ACCE55
        MCR     p14, 0, R1, c1, c0, 4           ; Write DBGOSLAR
        ISB

        ; (2) Get the number of words to save.
        MRC     p14, 0, R1, c1, c2, 4           ; DBGOSSRR returns size
        STR     R1, [R0], #4                    ; Push on to the save stack

        ; (3) Loop reading words from the DBGOSSRR.
        CMP     R1, #0                          ; Check for zero
SaveDebugRegisters_Loop
        ITTT    NE
        MRCNE   p14, 0, R2, c1, c2, 4           ; Load a word of data
        STRNE   R2, [R0], #4                    ; Push on to the save stack
        SUBSNE  R1, R1, #1
        BNE     SaveDebugRegisters_Loop

        ; (4) Return the pointer to first word not written to. This pointer is already in R0, so
        ;     all that is needed is to return from this function.
        ;
        ;     Leave DBGOSLAR set, because from now on we do not want any changes.
        BX      LR
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Example C6-4 OS debug register restore sequence, Extended CP14 interface

```
; On entry, R0 points to a block of saved debug registers.

RestoreDebugRegisters
        ; (1) Set OS Lock Access Register (DBGOSLAR) and reset pointer. The lock
        ;     will already be set, but this write is needed to reset the pointer.
        LDR     R1, =0xC5ACCE55
        MCR     p14, 0, R1, c1, c0, 4           ; Write DBGOSLAR
        ISB

        ; (2) Clear the Sticky Power-down Status bit.
        MRC     p14, 0, R1, c1, c5, 4           ; Read DBGPRSR to clear StickyPD
        ISB

        ; (3) Get the number of words saved.
        MRC     p14, 0, R1, c1, c2, 4           ; Dummy read of DBGOSSRR
        LDR     R1, [R0], #4                    ; Load size from the save stack

        ; (4) Loop writing words from the DBGOSSRR.
        CMP     R1, #0                          ; Check for zero
RestoreDebugRegisters_Loop
        ITTT    NE
        LDRNE   R2, [R0], #4                    ; Load a word from the save stack
        MCRNE   p14, 0, R2, c1, c2, 4           ; Store a word of data
        SUBSNE  R1, R1, #1
        BNE     RestoreDebugRegisters_Loop

        ; (5) Clear the OS Lock Access Register (DBGOSLAR). Writing any non-key value
        ;     clears the lock, so use the zero value in R1.
        ISB
        MCR     p14, 0, R1, c1, c0, 4           ; Write DBGOSLAR

        ; (6) A final ISB guarantees the restored register values are visible to subsequent
        ;     instructions.
        ISB

        ; (7) Return the pointer to first word not read. This pointer is already in R0, so
        ;     all that is needed is to return from this function.
        BX      LR
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

例 C6-4 OS デバッグレジスタの復元シーケンス、拡張 CP14 インタフェース


```
; On entry, R0 points to a block of saved debug registers.

RestoreDebugRegisters
        ; (1) Set OS Lock Access Register (DBGOSLAR) and reset pointer. The lock
        ;     will already be set, but this write is needed to reset the pointer.
        LDR     R1, =0xC5ACCE55
        MCR     p14, 0, R1, c1, c0, 4           ; Write DBGOSLAR
        ISB

        ; (2) Clear the Sticky Power-down Status bit.
        MRC     p14, 0, R1, c1, c5, 4           ; Read DBGPRSR to clear StickyPD
        ISB

        ; (3) Get the number of words saved.
        MRC     p14, 0, R1, c1, c2, 4           ; Dummy read of DBGOSSRR
        LDR     R1, [R0], #4                    ; Load size from the save stack

        ; (4) Loop writing words from the DBGOSSRR.
        CMP     R1, #0                          ; Check for zero
RestoreDebugRegisters_Loop
        ITTT    NE
        LDRNE   R2, [R0], #4                    ; Load a word from the save stack
        MCRNE   p14, 0, R2, c1, c2, 4           ; Store a word of data
        SUBSNE  R1, R1, #1
        BNE     RestoreDebugRegisters_Loop

        ; (5) Clear the OS Lock Access Register (DBGOSLAR). Writing any non-key value
        ;     clears the lock, so use the zero value in R1.
        ISB
        MCR     p14, 0, R1, c1, c0, 4           ; Write DBGOSLAR

        ; (6) A final ISB guarantees the restored register values are visible to subsequent
        ;     instructions.
        ISB

        ; (7) Return the pointer to first word not read. This pointer is already in R0, so
        ;     all that is needed is to return from this function.
        BX      LR
```


ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.2.4 Recommended reset scheme for v7 Debug

The processor reset scheme is IMPLEMENTATION DEFINED. The ARM architecture, described in parts A and B of this manual, does not distinguish different levels of reset. However, in a typical system, there are a number of reasons why multiple levels of reset might exist. In particular, for debug:

- It is desirable to be able to debug the reset sequence. This requires support for:
  - — setting the debug register values before performing a processor reset
  - — a processor reset not resetting the debug register values.
- Providing separate power domains means you might need to reset the debug logic independently from the logic in the core power domain.

For these reasons, v7 Debug introduces a distinction between *debug logic reset* and *non-debug logic reset*. These resets can be applied independently. The reset descriptions in parts A and B of this manual describe the non-debug logic reset. Part C describes the debug logic reset and its interaction with the non-debug logic reset. The non-debug logic reset is sometimes referred to as a *core logic reset*.

ARM recommends use of the following reset signals for an implementation that supports these independent resets:

nSYSPORESET This signal must be driven LOW on power-up of both the core and debug power domains. It sets parts of the processor logic, including debug logic, to a known state.

nCOREPORESET If the core power domain is powered down while the system is still powered up, this signal must be driven LOW when the core power domain is powered back up. It sets parts of the processor logic in the core power domain to a known state. Also, this reset initializes the debug registers that are in the core power domain.

nRESET This signal is driven LOW to generate a warm reset, that is, when the system wants to set the processor to a known state but the reset has nothing to do with any power-down, for example a watchdog reset. It sets parts of the non-debug processor logic to a known state. A debug session must be unaffected by this reset.

PRESETDBGn The debugger drives this signal LOW to set parts of the debug logic to a known state. This signal must be driven LOW on power-up of the debug logic.

v6 Debug and v6.1 Debug systems do not support multiple power domains and therefore ARM recommends a less flexible reset scheme, consisting of only **nSYSPORESET** and **nRESET**. The debug logic is reset only on **nSYSPORESET** and has no independent reset signal.

In the v7 Debug recommended reset scheme, a separate **PRESETDBGn** reset signal can be asserted at any time, not just at power-up. This new signal has similar effects to **nSYSPORESET**, that is, it clears all debug registers, unless otherwise noted by the register definition. For more information, see Appendix A *Recommended External Debug Interface*.

Asynchronously asserting **PRESETDBGn** can lead to UNPREDICTABLE behavior. For example, the reset might change the values of debug registers that are in use or will be used by software.

For more information about this reset scheme, contact ARM.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C6.2.4 v7 デバッグで推奨されるリセット方式

プロセッサのリセット方式は実装定義です。本書のパート A および B で説明した ARM アーキテクチャでは、異なるレベルのリセットが区別されていません。しかし、一般的なシステムでは、複数のレベルのリセットが存在する理由は数多くあります。デバッグに関しては、特に次の理由が当てはまります。

- リセットシーケンスのデバッグが可能であることが望ましい。このためには、次のサポートが必要となります。
  - – プロセッサのリセットを実行する前に、デバッグレジスタの値を設定する。
  - – プロセッサのリセットにより、デバッグレジスタの値がリセットされない。
- 独立した電力ドメインを持つ場合、デバッグロジックを、コア電力ドメインのロジックとは独立してリセットすることが必要な場合がある。

これらの理由から、v7 デバッグでは*デバッグロジック リセット*と*非デバッグロジック リセット*の区別が導入されています。これらのリセットは、独立に適用できます。本書のパート A および B では、非デバッグロジック リセットについて説明されています。パート C では、デバッグロジック リセットと、非デバッグロジック リセットとの相互動作について説明します。非デバッグロジック リセットは、*コアロジック リセット*と呼ばれることもあります。

これらの独立したリセットをサポートする実装では、次に示すリセット信号を使用することをお勧めします。

nSYSPORESET コアおよびデバッグ電力ドメイン両方の電力オン時に、この信号を LOW に駆動する必要があります。これによって、デバッグロジックを含むプロセッサロジックの一部が、既知の状態に設定されます。

nCOREPORESET システムが電力オンの動作中に、コア電力ドメインの電力がオフになった場合、コア電力ドメインの電力が再度オンになったときに、この信号を LOW に駆動する必要があります。これによって、コア電力ドメインにあるプロセッサロジックの一部が、既知の状態に設定されます。また、このリセットにより、コア電力ドメインにあるデバッグレジスタが初期化されます。

nRESET この信号は、ウォームリセットを生成する時、つまりシステムがプロセッサを既知の状態に設定する必要があるが、リセットが電力オフに関係のない場合、例えばウォッチドッグリセットの場合に LOW に駆動されます。これによって、非デバッグ部分のプロセッサロジックの一部が、既知の状態に設定されます。デバッグセッションは、このリセットの影響を受けない必要があります。

PRESETDBGn デバッガは、デバッグロジックの一部を既知の状態に設定するためにこの信号を LOW に駆動します。デバッグロジックの電力オン時には、この信号を LOW に駆動する必要があります。

v6 デバッグおよび v6.1 デバッグシステムは、複数の電力ドメインをサポートしていないため、nSYSPORESET と nRESET だけで構成される、柔軟性の低いリセット方式をお勧めします。デバッグロジックは nSYSPORESET でのみリセットされ、独立したリセット信号はありません。

v7 デバッグで推奨されるリセット方式では、独立した PRESETDBGn リセット信号を、電力オン時だけでなくいつでもアサートできます。この新しい信号は nSYSPORESET と同様の影響を持ち、レジスタ定義に特記されていない限り、すべてのデバッグレジスタをクリアします。詳細については、付録 A*「推奨される外部デバッグインタフェース」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C6-1 summarizes the v7 Debug recommended reset scheme.

Table C6-1 Recommended reset scheme, v7 Debug

	Debug power domain	Core power domain	
Signal	**Debug logic**	**Debug logic**	**Non-debug logic**
nSYSPORESET	Reset	Reset[a]	Reset
nCOREPORESET	Not reset	Reset[a]	Reset
nRESET	Not reset	Not reset	Reset
PRESETDBGn	Reset	Reset[a]	Not reset

a. If the core power domain is not powered, or the Sticky Power-down status bit **DBGPRSR[1]** is set to 1, it is UNPREDICTABLE whether the registers are reset. If power is not applied to the core power domain, **nCOREPORESET** must be driven LOW when power is restored to the core power domain. This resets these registers.

For ARMv7 SinglePower systems, ARM recommends only **nSYSPORESET**, **nRESET**, and **PRESETDBGn**.

Debug behavior when the processor is in debug logic reset

The implementation of separate debug and core power domains with a separate debug logic reset signal means that a processor can access debug registers and the DCC while in the debug logic reset state. When in debug logic reset:

- The behavior of the DCC is UNPREDICTABLE. In particular, the values of the DBGDSCR.RXfull and DBGDSCR.TXfull flags are UNKNOWN.
- It is UNPREDICTABLE whether a debug event that would have been generated by the state of the debug logic immediately before the debug logic reset is generated.
- The debug logic must not generate any debug event that would not have been generated if the system was not in debug logic reset.
- Accesses to the debug registers through the Extended CP14, memory-mapped and external debug interfaces are UNPREDICTABLE.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

PRESETDBGn を非同期にアサートすると、予測不能な動作を引き起こすことがあります。例えば、リセットによって、ソフトウェアにより使用中、または使用される予定のデバッグレジスタの値が変化する可能性があります。

このリセット方式の詳細については、ARM にお問い合わせ下さい。

v7 デバッグで推奨されるリセット方式の概要を、表 C6-1 に示します。

表 C6-1 推奨されるリセット方式、v7 デバッグ

	デバッグ電力ドメイン	コア電力ドメイン	
信号	**デバッグロジック**	**デバッグロジック**	**非デバッグロジック**
nSYSPORESET	リセット	リセット[a]	リセット
nCOREPORESET	リセットされない	リセット[a]	リセット
nRESET	リセットされない	リセットされない	リセット
PRESETDBGn	リセット	リセット[a]	リセットされない

a. コア電力ドメインの電力がオンになっていない場合、またはスティッキー電力オフステータスビット (DBGPRSR[1]) が 1 にセットされている場合、このレジスタがリセットされるかどうかは予測不能です。コア電力ドメインに電力が供給されていない場合、コア電力ドメインの電力が回復したときに nCOREPORESET を LOW に駆動する必要があります。これにより、これらのレジスタがリセットされます。

ARMv7 の SinglePower システムについては、nSYSPORESET、nRESET、PRESETDBGn のみを推奨します。

プロセッサがデバッグロジック リセット中のデバッグ動作

独立したデバッグとコアの電力ドメインを実装し、独立したデバッグロジック リセット信号を持つことは、デバッグロジックのリセット状態の間にプロセッサがデバッグレジスタと DCC にアクセスできることを意味します。デバッグロジックのリセット中は、次の規則が適用されます。

- DCC の動作は予測不能です。特に、DBGDSCR.RXfull および DBGDSCR.TXfull フラグの値は未知です。
- デバッグロジック リセット直前のデバッグロジックの状態によって生成されるべきデバッグイベントが、生成されるかどうかは予測不能です。
- デバッグロジックは、システムがデバッグロジック リセット中でないと仮定した場合に生成されないような、デバッグイベントを生成することはできません。
- 拡張 CP14、メモリマップされたインタフェース、および外部デバッグインタフェース経由でのデバッグレジスタへのアクセスは予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.3 Debug register map

Table C6-2 lists all of the debug registers. Full details of each register can be found in the referenced section.

The number of DBGBVR/DBGBCR and DBGWVR/DBGWCR pairs is IMPLEMENTATION DEFINED, see the BRPs and WRPs fields of the *Debug ID Register (DBGDIDR)* on page C10-3. An implementation can have up to 16 of each.

The interpretation of the information in the *Access* column depends on whether the coprocessor or memory-mapped interface is used to access the register.

Collectively, registers 832-1023 are known as the management registers.

Table C6-2 Debug register map

Register number	Offset	Access[a]	Versions[b]	Name and reference to description
0	0x000	Read-only	All	*Debug ID Register (DBGDIDR)* on page C10-3.
Not applicable[c]	-	Read-only	v7 only	*Debug ROM Address Register (DBGDRAR)* on page C10-7.
Not applicable[c]	-	Read-only	v7 only	*Debug Self Address Offset Register (DBGDSAR)* on page C10-8.
1-5	-	-	-	Reserved.
6	0x018	Read/write	v7 [d]	*Watchpoint Fault Address Register (DBGWFAR)* on page C10-28.
7	0x01C	Read/write	All	*Vector Catch Register (DBGVCR)* on page C10-67.
8	-	-	-	Reserved.
9	0x024	Read/write	v7 only	*Event Catch Register (DBGECR)* on page C10-78.
10	0x028	Read/write	v6.1, v7	*Debug State Cache Control Register (DBGDSCCR)* on page C10-81.
11	0x02C	Read/write	v6.1, v7	*Debug State MMU Control Register (DBGDSMCR)* on page C10-84.
12-31	-	-	-	Reserved.
32	0x080	Read/write	v7 [e]	DBGDTRRX external view[f]. See *Host to Target Data Transfer Register (DBGDTRRX)* on page C10-40.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C6.3 デバッグレジスタのマップ

すべてのデバッグレジスタの一覧を、表 C6-2 に示します。各レジスタの完全な詳細については、指定されたセクションを参照して下さい。

DBGBVR/DBGBCR および DBGWVR/DBGWCR ペアの数は実装定義です。*デバッグ ID レジスタ (DBGDIDR)* P. C10-3 の BRP および WRP のフィールドを参照して下さい。実装にはそれぞれを最大 16 個まで含めることができます。

*アクセス*列にある情報の解釈は、そのレジスタにアクセスするためにコプロセッサやメモリマップされたインタフェースが使用されるかどうかによって異なります。

レジスタ 832 ～ 1023 は、まとめて管理レジスタと呼ばれます。

表 C6-2 デバッグレジスタのマップ

レジスタ番号	オフセット	アクセス[a]	バージョン[b]	名前および説明されているセクション
0	0x000	読み出し専用	すべて	*デバッグ ID レジスタ (DBGDIDR)* P. C10-3
該当せず[c]	–	読み出し専用	v7 のみ	*デバッグ ROM アドレスレジスタ (DBGDRAR)* P. C10-7
該当せず[c]	–	読み出し専用	v7 のみ	*デバッグ自己アドレスオフセット レジスタ (DBGDSAR)* P. C10-8
1 ～ 5	–	–	–	予約
6	0x018	読み出し / 書き込み	v7[d]	*ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10-28
7	0x01C	読み出し / 書き込み	すべて	*ベクタキャッチ レジスタ (DBGVCR)* P. C10-67
8	–	–	–	予約
9	0x024	読み出し / 書き込み	v7 のみ	*イベントキャッチ レジスタ (DBGECR)* P. C10-78
10	0x028	読み出し / 書き込み	v6.1、v7	*デバッグ状態キャッシュ制御レジスタ (DBGDSCCR)* P. C10-81
11	0x02C	読み出し / 書き込み	v6.1、v7	*デバッグ状態 MMU 制御レジスタ (DBGDSMCR)* P. C10-84
12 ～ 31	–	–	–	予約
32	0x080	読み出し / 書き込み	v7[e]	DBGDTRRX の外部ビュー[f]。*ホストからターゲットへのデータ転送レジスタ (DBGDTRRX)* P. C10-40 を参照して下さい。
33	0x084	書き込み専用	v7[e]	*命令転送レジスタ (DBGITR)* P. C10-46
		読み出し専用	v7[e]	*プログラムカウンタサンプリング レジスタ (DBGPCSR)* P. C10-38

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C6-2 Debug register map (continued)

Register number	Offset	Access[a]	Versions[b]	Name and reference to description
33	0x084	Write-only	v7 [e]	*Instruction Transfer Register (DBGITR)* on page C10-46.
		Read-only	v7 [e]	*Program Counter Sampling Register (DBGPCSR)* on page C10-38.
34	0x088	Read/write	v7 [e]	DBGDSCR external view[f]. See *Debug Status and Control Register (DBGDSCR)* on page C10-10.
35	0x08C	Read/write	v7 [e]	DBGDTRTX external view[f]. See *Target to Host Data Transfer Register (DBGDTRTX)* on page C10-43.
36	0x090	Write-only	v7 only	*Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29.
37-39	-	-	-	Reserved.
40	0x0A0	Read-only	v7 only	*Program Counter Sampling Register (DBGPCSR)* on page C10-38
41	0x0A4	Read-only	v7 only	*Context ID Sampling Register (DBGCIDSR)* on page C10-39
42-63	-	-	-	Reserved.
64-79	0x100-0x13C	Read/write or -	All	*Breakpoint Value Registers (DBGBVR)* on page C10-48 or Reserved.
80-95	0x140-0x17C	Read/write or -	All	*Breakpoint Control Registers (DBGBCR)* on page C10-49 or Reserved.
96-111	0x180-0x1BC	Read/write or -	All	*Watchpoint Value Registers (DBGWVR)* on page C10-60 or Reserved.
112-127	0x1C0-0x1FC	Read/write or -	All	*Watchpoint Control Registers (DBGWCR)* on page C10-61 or Reserved.
128-191	-	-	-	Reserved.
192	0x300	Write-only	v7 only	*OS Lock Access Register (DBGOSLAR)* on page C10-75.
193	0x304	Read-only	v7 only	*OS Lock Status Register (DBGOSLSR)* on page C10-76.
194	0x308	Read/write	v7 only	*OS Save and Restore Register (DBGOSSRR)* on page C10-77.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C6-2 デバッグレジスタのマップ（続き）

レジスタ番号	オフセット	アクセス[a]	バージョン[b]	名前および説明されているセクション
34	0x088	読み出し / 書き込み	v7[e]	DBGDSCR の外部ビュー[f]。*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。
35	0x08C	読み出し / 書き込み	v7[e]	DBGDTRTX の外部ビュー[f]。*ターゲットからホストへのデータ転送レジスタ (DBGDTRTX)* P. C10-43 を参照して下さい。
36	0x090	書き込み専用	v7 のみ	*デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10-29
37 ～ 39	–	–	–	予約
40	0x0A0	読み出し専用	v7 のみ	*プログラムカウンタサンプリング レジスタ (DBGPCSR)* P. C10-38
41	0x0A4	読み出し専用	v7 のみ	*コンテキスト ID サンプリングレジスタ (DBGCIDSR)* P. C10-39
42 ～ 63	–	–	–	予約
64 ～ 79	0x100 ～ 0x13C	読み出し / 書き込み または –	すべて	*ブレークポイント値レジスタ (DBGBVR)* P. C10-48 または予約
80 ～ 95	0x140 ～ 0x17C	読み出し / 書き込み または –	すべて	*ブレークポイント制御レジスタ (DBGBCR)* P. C10-49 または予約
96 ～ 111	0x180 ～ 0x1BC	読み出し / 書き込み または –	すべて	*ウォッチポイント値レジスタ (DBGWVR)* P. C10-60 または予約
112 ～ 127	0x1C0 ～ 0x1FC	読み出し / 書き込み または –	すべて	*ウォッチポイント制御レジスタ (DBGWCR)* P. C10-61 または予約
128 ～ 191	–	–	–	予約
192	0x300	書き込み専用	v7 のみ	*OS ロックアクセス レジスタ (DBGOSLAR)* P. C10-75
193	0x304	読み出し専用	v7 のみ	*OS ロックステータス レジスタ (DBGOSLSR)* P. C10-76
194	0x308	読み出し / 書き込み	v7 のみ	*OS 保存および復元レジスタ (DBGOSSRR)* P. C10-77
195	–	–	–	予約
196	0x310	読み出し / 書き込み	v7 のみ	*デバイス電力オフおよびリセット制御レジスタ (DBGPRCR)、v7 デバッグのみ* P. C10-31

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C6-2 Debug register map (continued)

Register number	Offset	Access[a]	Versions[b]	Name and reference to description
195	-	-	-	Reserved.
196	0x310	Read/write	v7 only	*Device Power-down and Reset Control Register (DBGPRCR), v7 Debug only* on page C10-31.
197	0x314	Read-only	v7 only	*Device Power-down and Reset Status Register (DBGPRSR), v7 Debug only* on page C10-34.
198-511	-	-	-	Reserved.
512-575	0x800-0x8FC	-	v7 only	IMPLEMENTATION DEFINED.
576-831	-	-	-	Reserved.
832-895	0xD00-0xDFC	Read-only	v7 only	*Processor identification registers* on page C10-88.
896-927	-	-	-	Reserved.
928-959	0xE80-0xEFC	-	v7 only	IMPLEMENTATION DEFINED integration registers. See the *CoreSight Architecture Specification*.
960	0xF00	Read/write	v7 only	*Integration Mode Control Register (DBGITCTRL)* on page C10-91.
961-999	0xF04-0xF9C	-	v7 only	Reserved for management registers expansion.
1000	0xFA0	Read/write	v7 only	*Claim Tag Set Register (DBGCLAIMSET)* on page C10-92.
1001	0xFA4	Read/write	v7 only	*Claim Tag Clear Register (DBGCLAIMCLR)* on page C10-93.
1002-1003	-	-	-	Reserved.
1004	0xFB0	Write-only	v7 only	*Lock Access Register (DBGLAR)* on page C10-94.
1005	0xFB4	Read-only	v7 only	*Lock Status Register (DBGLSR)* on page C10-95.
1006	0xFB8	Read-only	v7 only	*Authentication Status Register (DBGAUTHSTATUS)* on page C10-96.
1007-1009	-	-	-	Reserved.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 C6-2 デバッグレジスタのマップ（続き）

レジスタ番号	オフセット	アクセス[a]	バージョン[b]	名前および説明されているセクション
197	0x314	読み出し専用	v7 のみ	*デバイス電力オフおよびリセットステータス レジスタ (DBGPRSR)、v7 デバッグのみ* P. C10-34
198 ～ 511	–	–	–	予約
512 ～ 575	0x800 ～ 0x8FC	–	v7 のみ	実装定義
576 ～ 831	–	–	–	予約
832 ～ 895	0xD00 ～ 0xDFC	読み出し専用	v7 のみ	*プロセッサ識別レジスタ* P. C10-88
896 ～ 927	–	–	–	予約
928 ～ 959	0xE80 ～ 0xEFC	–	v7 のみ	実装定義の統合レジスタ。『*CoreSight アーキテクチャ仕様*』を参照して下さい。
960	0xF00	読み出し / 書き込み	v7 のみ	*統合モード制御レジスタ (DBGITCTRL)* P. C10-91
961 ～ 999	0xF04 ～ 0xF9C	–	v7 のみ	管理レジスタの拡張用に予約済み
1000	0xFA0	読み出し / 書き込み	v7 のみ	*クレームタグ セットレジスタ (DBGCLAIMSET)* P. C10-92
1001	0xFA4	読み出し / 書き込み	v7 のみ	*クレームタグ クリアレジスタ (DBGCLAIMCLR)* P. C10-93
1002 ～ 1003	–	–	–	予約
1004	0xFB0	書き込み専用	v7 のみ	*ロックアクセス レジスタ (DBGLAR)* P. C10-94
1005	0xFB4	読み出し専用	v7 のみ	*ロックステータス レジスタ (DBGLSR)* P. C10-95
1006	0xFB8	読み出し専用	v7 のみ	*認証ステータスレジスタ (DBGAUTHSTATUS)* P. C10-96
1007 ～ 1009	–	–	–	予約
1010	0xFC8	読み出し専用	v7 のみ	*デバッグデバイス ID レジスタ (DBGDEVID)* P. C10-6

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C6-2 Debug register map (continued)

Register number	Offset	Access[a]	Versions[b]	Name and reference to description
1010	0xFC8	Read-only	v7 only	*Debug Device ID Register (DBGDEVID)* on page C10-6.
1011	0xFCC	Read-only	v7 only	*Device Type Register (DBGDEVTYPE)* on page C10-98.
1012-1019	0xFD0-0xFEC	Read-only	v7 only	*Debug Peripheral Identification Registers (DBGPID0 to DBGPID4)* on page C10-98.
1020-1023	0xFF0-0xFFC	Read-only	v7 only	*Debug Component Identification Registers (DBGCID0 to DBGCID3)* on page C10-102.

a. For more information, see *CP14 debug registers access permissions* on page C6-36 and *Permission summaries for memory-mapped and external debug interfaces* on page C6-45.
b. An entry of *All* in the *Versions* column indicates that the register is implemented in v6 Debug, v6.1 Debug, and v7 Debug.
c. These registers are only implemented through the Baseline CP14 interface and do not have register numbers or offsets.
d. The method of accessing the DBGWFAR is different in v6 Debug, v6.1 Debug and v7 Debug. For details see *Watchpoint Fault Address Register (DBGWFAR)* on page C10-28.
e. In v6 Debug and v6.1 Debug, ARM recommends these registers as part of the external debug interface, and are not implemented through the Extended CP14 interface. In v7 Debug these registers are required.
f. Internal views of the DBGDTRRX, DBGDTRTX, and DBGDSCR are implemented through the Baseline CP14 interface. This is explained in *Internal and external views of the DBGDSCR and the DCC registers*.

C6.3.1 Internal and external views of the DBGDSCR and the DCC registers

For each of the three registers DBGDSCR, DBGDTRTX and DBGDTRRX there are two views, denoted by int and ext suffixes. The differences between these aliases relate to the handling of the *Debug Communications Channel* (DCC), and in particular the TXfull and RXfull status flags. The nomenclature *internal* and *external* derives from the intended usage model.

Accesses to DBGDSCRint, DBGDTRRXint or DBGDTRTXint are always made through the Baseline CP14 interface described in *The Baseline CP14 debug register interface* on page C6-32. DBGDSCRint is read-only in v7 Debug.

Accesses to DBGDSCRext, DBGDTRRXext or DBGDTRTXext can be made through:

- the Extended CP14 interface, if implemented
- the memory-mapped interface, if implemented
- the external debug interface.

However, if at any given time you attempt to access the DBGDSCRext, DBGDTRRXext and DBGDTRTXext registers through more than one interface the behavior is UNPREDICTABLE. If an implementation provides a single port to handle external debug interface and the memory-mapped interface

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C6-2 デバッグレジスタのマップ（続き）

レジスタ番号	オフセット	アクセス[a]	バージョン[b]	名前および説明されているセクション
1011	0xFCC	読み出し専用	v7 のみ	*デバイスタイプ レジスタ (DBGDEVTYPE)* P. C10-98
1012 ～ 1019	0xFD0 ～ 0xFEC	読み出し専用	v7 のみ	*デバッグペリフェラル識別レジスタ (DBGPID0 ～ DBGPID4)* P. C10-98
1020 ～ 1023	0xFF0 ～ 0xFFC	読み出し専用	v7 のみ	*デバッグコンポーネント識別レジスタ (DBGCID0 ～ DBGCID3)* P. C10-102

a. 詳細については、*CP14 デバッグレジスタのアクセス許可* P. C6-36 と *メモリマップおよび外部デバッグインタフェースのアクセス許可の概要* P. C6-45 を参照して下さい。
b. *バージョン*列ですべてと記載されているエントリは、そのレジスタが v6 デバッグ、v6.1 デバッグ、v7 デバッグのすべてに実装されることを意味します。
c. これらのレジスタは、ベースライン CP14 インタフェース経由でのみ実装され、レジスタ番号やオフセットは存在しません。
d. DBGWFAR にアクセスする手法は、v6 デバッグ、v6.1 デバッグ、v7 デバッグでそれぞれ異なります。詳細については、*ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10-28 を参照して下さい。
e. v6 デバッグおよび v6.1 デバッグでは、これらのレジスタを外部デバッグインタフェースの一部とし、拡張 CP14 インタフェース経由では実装しないことをお勧めします。v7 デバッグでは、これらのレジスタは必須です。
f. DBGDTRRX、DBGDTRTX、DBGDSCR は、ベースライン CP14 インタフェース経由で実装されます。説明については、*DBGDSCR および DCC レジスタの内部および外部ビュー*を参照して下さい。

C6.3.1 DBGDSCR および DCC レジスタの内部および外部ビュー

DBGDSCR、DBGDTRTX、DBGDTRRX の 3 つのレジスタには、それぞれ 2 つのビューがあり、int および ext という接尾辞により示されます。これらのエイリアス間の相違点は、*デバッグ通信チャネル*(DCC) の扱い、特に TXfull および RXfull ステータスフラグの扱いに関連しています。*内部*および*外部*という用語は、意図された使用モデルに基づいています。

DBGDSCRint、DBGDTRRXint、DBGDTRTXint へのアクセスは、常に*ベースライン CP14 デバッグレジスタ インタフェース* P. C6-32 で説明されているベースライン CP14 インタフェース経由で行われます。DBGDSCRint は、v7 デバッグでは読み出し専用です。

DBGDSCRext、DBGDTRRXext、DBGDTRTXext へのアクセスは、次のインタフェースを経由して行われます。

- 拡張 CP14 インタフェース（実装されている場合）
- メモリマップされたインタフェース（実装されている場合）
- 外部デバッグインタフェース

ただし、ある時点において、DBGDSCRext、DBGDTRRXext、DBGDTRTXext レジスタに複数のインタフェースを経由してアクセスを試みた場合、動作は予測不能です。実装で、外部デバッグインタフェースとメモリマップされたインタフェースのアクセスを単一のポートで処理している場合、そのポートはこれら 2 つのインタフェースからレジスタへのアクセスを直列化する必要があります。ただし、これらのレジスタへの読み出しおよび書き込みの影響は、いずれのインタフェースから観測しても動作が予測不能となります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

accesses, that port might serialize accesses to the registers from the two interfaces. However, the effects of reads and writes to these registers are such that the behavior observed from either interface appears as UNPREDICTABLE.

Figure C6-2 v7 Debug Internal (int) and External (ext) views of the DCC registers

Note

- DBGDSCRint and DBGDSCRext only provide different views onto the underlying DBGDSCR
- DBGDTRRXint and DBGDTRRXext only provide different views onto the underlying DBGDTRRX Register
- DBGDTRTXint and DBGDTRTXext only provide different views onto the underlying DBGDTRTX Register.

See also:

- *Debug Status and Control Register (DBGDSCR)* on page C10-10
- *Host to Target Data Transfer Register (DBGDTRRX)* on page C10-40
- *Target to Host Data Transfer Register (DBGDTRTX)* on page C10-43.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

図 C6-2 v7 デバッグでの DCC レジスタの内部 (int) および外部 (ext) ビュー

注

- DBGDSCRint と DBGDSCRext は、基礎となる DBGDSCR の異なるビューを提供するだけです。
- DBGDTRRXint と DBGDTRRXext は、基礎となる DBGDTRRX レジスタの異なるビューを提供するだけです。
- DBGDTRTXint と DBGDTRTXext は、基礎となる DBGDTRTX レジスタの異なるビューを提供するだけです。

次のセクションも参照して下さい。

- *デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10
- *ホストからターゲットへのデータ転送レジスタ (DBGDTRRX)* P. C10-40
- *ターゲットからホストへのデータ転送レジスタ (DBGDTRTX)* P. C10-43

C6.3.2 セキュリティ拡張機能のデバッグレジスタへの影響

セキュリティ拡張機能が実装されている場合は、すべてのデバッグレジスタが共通レジスタで、セキュアと非セキュア両方の状態で共通です。詳細については、*共通の CP15 レジスタ* P. B3-74 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.3.2 Effect of the Security Extensions on the debug registers

When the Security Extensions are implemented, all debug register are Common registers, meaning they are common to the Secure and Non-secure states. For more information, see *Common CP15 registers* on page B3-74.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.4 デバッグレジスタの更新の同期

プロセッサで実行されているソフトウェアは、次のうち最低 1 つを使用してデバッグレジスタをプログラムできます。

- CP14 コプロセッサインタフェース
- メモリマップされたインタフェース（実装されている場合）

どのインタフェースが実装されているかは実装定義です。

CP14 コプロセッサインタフェースの場合、次の同期規則が適用されます。

- プログラム順序で、任意の明示的メモリ操作の後に出現する CP14 デバッグレジスタの変更は、そのメモリ操作に影響を及ぼさないことが保証されます。
- CP14 デバッグレジスタの変更は、次のいずれかが発生した後でのみ、以後の命令から可視であることが保証されます。
  - ISB 操作の実行
  - 例外の取得
  - 例外からの復帰

  ただし、CP14 コプロセッサレジスタへのアクセスの場合、同じレジスタ番号を使用した同じレジスタへの MRC および MCR 命令はすべて、互いに相対的にプログラム上の順序で、コンテキスト同期なしに発生したように見えます。

メモリマップされたインタフェースの場合、次の同期規則が適用されます。

- すべてのメモリマップされたデバッグレジスタは、ストロングリオーダまたはデバイスメモリにマッピングされる必要があります。そうでない場合、メモリマップされたデバッグレジスタへのアクセスの影響は予測不能です。
- メモリマップされたデバッグレジスタの変更が、プログラムの順序で明示的なメモリ操作の後に出現する場合、その順序がメモリオーダ モデルにより保証されているか、メモリ操作とレジスタ変更との間に DMB または DSB 操作が使用されている場合のみ、デバッグレジスタの変更が前のメモリ操作に影響しないことが保証されます。
- DSB 操作は、プログラムの順序でその DSB の前に出現する、すべてのメモリマップされたデバッグレジスタへの書き込みを完了させます。
- 同じプロセッサによる、メモリマップされたデバッグレジスタへの他のアクセスに関して、メモリマップされたデバッグレジスタへのすべてのアクセスは、メモリオーダ モデルと DSB および DMB 操作の使用による調整に従い、アクセスが発生する順序で影響を及ぼします。
- メモリマップされたデバッグレジスタへのすべての完了したアクセスは、次のいずれかが行われた後でのみ、以後の命令に影響することが保証されます。
  - ISB 操作の実行
  - 例外の取得
  - 例外からの復帰

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.4 Synchronization of debug register updates

Software running on the processor can program the debug registers through at least one of:

- a CP14 coprocessor interface
- the memory-mapped interface, if it is implemented.

It is IMPLEMENTATION DEFINED which interfaces are implemented.

For the CP14 coprocessor interface, the following synchronization rules apply:

- All changes to CP14 debug registers that appear in program order after any explicit memory operations are guaranteed not to affect those memory operations.
- Any change to CP14 debug registers is guaranteed to be visible to subsequent instructions only after one of:
  - — performing an ISB operation
  - — taking an exception
  - — returning from an exception.

  However, for CP14 coprocessor register accesses, all `MRC` and `MCR` instructions to the same register using the same register number appear to occur in program order relative to each other without context synchronization.

For the memory-mapped interface, the following synchronization rules apply:

- All memory-mapped debug registers must be mapped to Strongly-ordered or Device memory, otherwise the effect of any access to the memory-mapped debug registers is `UNPREDICTABLE`.
- Changes to memory-mapped debug registers that appear in program order after an explicit memory operation are guaranteed not to affect that previous memory operation only if the order is guaranteed by the memory order model or by the use of a DMB or DSB operation between the memory operation and the register change.
- A DSB operation causes all writes to memory-mapped debug registers appearing in program order before the DSB to be completed.
- With respect to other accesses by the same processor to the memory-mapped debug registers, all accesses to memory-mapped debug registers have their effect in the order in which the accesses occur, as governed by the memory order model and the use of DSB and DMB operations.
- All accesses to memory-mapped debug registers that are completed are only guaranteed to affect subsequent instructions after one of:
  - — performing an ISB operation
  - — taking an exception
  - — returning from an exception.

Some memory-mapped debug registers are not idempotent for reads or writes. Therefore, the region of memory occupied by the debug registers must not be marked as Normal memory, because the memory order model permits accesses to Normal memory locations that are not appropriate for such registers.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

一部のメモリマップされたデバッグレジスタは、読み出しまたは書き込みについてベキ等ではありません。このため、デバッグレジスタによって占有される領域のメモリを通常メモリとしてマークすることはできません。これは、メモリオーダ モデルにおいて、通常メモリの位置に対して、このようなレジスタに適切でないアクセスが許可されるためです。

外部デバッグインタフェース経由で行われたレジスタ更新と、プロセッサで実行されているソフトウェアにより行われた更新との同期は実装定義です。ただし、外部デバッグインタフェースがメモリマップされたインタフェースと同じポート経由で実装されている場合、外部デバッグインタフェース経由で行われた更新と、メモリマップされたインタフェース経由で行われた更新とのプロパティは同じです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Synchronization between register updates made through the external debug interface and updates made by software running on the processor is IMPLEMENTATION DEFINED. However, if the external debug interface is implemented through the same port as the memory-mapped interface, then updates made through the external debug interface have the same properties as updates made through the memory-mapped interface.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.5 アクセス許可

このセクションでは、ARMv7 プロセッサのデバッグレジスタのアクセス許可モデルに関する基本的な概念について説明します。各インタフェースの実際の規則、および ARMv6 実装でのモデルについては、レジスタインタフェースについて説明されている次のセクションを参照して下さい。

- *CP14 デバッグレジスタのアクセス許可* P. C6-36
- *メモリマップおよび外部デバッグインタフェースのアクセス許可の概要* P. C6-45

レジスタへのアクセスに関する制限は、次の 3 つのカテゴリに分けられます。

アクセスの特権

システムのプロセッサからメモリマップされたレジスタへのアクセスと、コプロセッサレジスタへのアクセスは、特権アクセスが必要な場合があります。

ロック

レジスタマップの各部分をロックし、アクセスできないようにするために使用できます。

電力オフ

コア電力ドメインの電力がオフになっている間は、そのドメインのレジスタにはアクセスできません。

レジスタにアクセスする許可が与えられない場合、エラーが返されます。このエラーの性質は、インタフェースによって異なります。

- コプロセッサインタフェースの場合、このエラーは未定義命令例外です。
- メモリマップされたインタフェースの場合、このエラーはスレーブによって生成されたエラー応答、例えば PSLVERRDBG です。このエラーは、通常は外部アボートとしてプロセッサに通知されます。
- 外部デバッグインタフェースの場合、このエラーはデバッグアクセス ポートによりデバッガに通知されます。

外部のウォームリセット信号またはデバイス電力オフおよびリセット制御レジスタ (DBGPRCR) を使用してプロセッサをウォームリセットに保持しても、メモリマップまたは外部デバッグインタフェースの動作には影響しません。

DBGPRCR の非デバッグリセット保持制御ビットにより、外部デバッガは他のデバッグレジスタをプログラムする間、プロセッサをウォームリセットに保持できます。詳細については、*デバイス電力オフおよびリセット制御レジスタ (DBGPRCR)、v7 デバッグのみ* P. C10-31 を参照して下さい。

C6.5.1 アクセスの特権とアクセス許可との関係

デバッグレジスタの大部分は、特権コードによってのみアクセス可能です。この一般的な要件の例外は、*ベースライン CP14 デバッグレジスタ インタフェース* P. C6-32 で定義されているレジスタの小さなサブセットです。特権コードからコプロセッサインタフェースを使用して、このレジスタのサブセットに対して、ユーザモードのアクセスを不可能にすることもできます。

メモリマップされたインタフェースの場合、デバッグレジスタへのアクセスを特権コードに制限する機構がプロセッサにより実装されるか、システム設計者がシステムレベルで実装する必要があるかは実装定義です。許可されていないアクセスの動作は実装定義ですが、無視またはアボートの必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.5 Access permissions

This section describes the basic concepts of the access permissions model for debug registers on ARMv7 processors. The actual rules for each interface, and for ARMv6 implementations, are given in the section describing the register interface:

- *CP14 debug registers access permissions* on page C6-36
- *Permission summaries for memory-mapped and external debug interfaces* on page C6-45.

The restrictions for accessing the registers can be divided into three categories:

Privilege of the access
Accesses from processors in the system to the memory-mapped registers, and accesses to coprocessor registers, can be required to be privileged.

Locks Can be used to lock out different parts of the register map so they cannot be accessed.

Power-down Access to registers in the core power domain is not possible when that domain is powered down.

When permission to access a register is not granted, an error is returned. The nature of this error depends on the interface:

- For coprocessor interfaces, the error is an Undefined Instruction exception
- For the memory-mapped interface, the error is a slave-generated error response, for example **PSLVERRDBG**. The error is normally signaled to the processor as an external abort.
- For the external debug interface, the error is signaled to the debugger by the Debug Access Port.

Holding the processor in warm reset, whether by using an external warm reset signal or by using the Device Power-down and Reset Control Register (DBGPRCR), does not affect the behavior of the memory-mapped or external debug interface.

The Hold non-debug reset control bit of the DBGPRCR enables an external debugger to keep the processor in warm reset while programming other debug registers. For details see *Device Power-down and Reset Control Register (DBGPRCR), v7 Debug only* on page C10-31.

C6.5.1 Permissions in relation to the privilege of the access

The majority of debug registers can only be accessed by privileged code. The exception to this general requirement is a small subset of the registers, defined in *The Baseline CP14 debug register interface* on page C6-32. Using the coprocessor interface, privileged code can disable User mode access to this subset of registers.

For the memory-mapped interface, it is IMPLEMENTATION DEFINED whether restricting debug register access to privileged code is implemented by the processor or must be implemented by the system designer at the system level. The behavior of an access that is not permitted is IMPLEMENTATION DEFINED, however it must either be ignored or aborted.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

- 推奨されるメモリマップされたインタフェースポートは、AMBA® アドバンスト ペリフェラルバス (APBv3) に基づいたもので、アクセス特権の通知をサポートしていません。したがって、この場合にはシステムがアクセスを防止する必要があります。
- このアクセス制限は、アクセスのイニシエータの特権に適用され、アクセスされるプロセッサの現在のモードは関係しません。デバッグアクセス ポートにより行われるアクセスの特権は実装定義です。

システム設計者が追加の制限を設けることもできます。ただし、セキュア特権アクセスでのアクセスのみを許可するなどの制限を設けないことと、そのような制限をデバッグツールでサポートしないことを強く推奨します。

C6.5.2 ロックとアクセス許可との関係

レジスタをデバッガまたはオペレーティングシステムによりロックし、デバッグレジスタへのアクセスを制限することができます。

ロックには 3 種類ありますが、ロックのいくつかは特定のインタフェースにのみ適用されます。

ソフトウェアロック
ソフトウェアロックは、メモリマップされたインタフェース経由で行われるアクセスにのみ適用されます。

デフォルトでは、ソフトウェアはロックされ、デバッグレジスタの設定は変更できません。デバッグモニタは、エラーのあるコードによってデバッグ設定が変更される可能性を減らすため、デバッグレジスタにアクセスしないときはこのロックを設定されたままにしておく必要があります。このロックが設定されている場合、メモリマップされたインタフェースからデバッグレジスタへの書き込みは無視されます。このロックの詳細については、*ロックアクセス レジスタ (DBGLAR)* P. C10-94 および*ロックステータス レジスタ (DBGLSR)* P. C10-95 を参照して下さい。

OS ロック OS は、OS 保存または復元シーケンスを開始する前に、デバッグレジスタにこのロックを設定し、シーケンスの間にデバッグレジスタの読み出しまたは書き込みが行われないようにする必要があります。このロックが設定されている場合、一部のレジスタへのアクセスでエラーが返されます。OS 保存および復元機構のレジスタのみが、安全にアクセス可能です。

注

外部デバッガは、たとえ OS 保存または復元操作が進行中でも、いつでもこのロックをクリアできます。

このロックの詳細については、*OS ロックアクセス レジスタ (DBGOSLAR)* P. C10-75 および *OS ロックステータス レジスタ (DBGOSLSR)* P. C10-76 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

- The recommended memory-mapped interface port is based on the AMBA® Advanced Peripheral Bus (APBv3), that does not support signaling of access privileges. Therefore in this case the system must prevent the access.
- This access restriction applies to the privilege of the initiator of the access, not the current mode of the processor being accessed. The privilege of accesses made by a Debug Access Port is IMPLEMENTATION DEFINED.

The system designer can impose additional restrictions. However, ARM strongly recommends that designers do not impose restrictions such as only permitting Secure privileged accesses, and does not support such restrictions in its debug tools.

C6.5.2 Permissions in relation to locks

The registers can be locked by a debugger or by an operating system so that access to debug registers is restricted.

There are three locks, although some of these locks only apply to certain interfaces:

Software Lock

The Software Lock only applies to accesses made through the memory-mapped interface.

By default, software is locked out so the debug registers settings cannot be modified. A debug monitor must leave this lock set when not accessing the debug registers, to reduce the chance of erratic code modifying debug settings. When this lock is set, writes to the debug registers from the memory-mapped interface are ignored. For more information about this lock, see *Lock Access Register (DBGLAR)* on page C10-94 and *Lock Status Register (DBGLSR)* on page C10-95.

OS Lock An OS must set this lock on the debug registers before starting an OS Save or Restore sequence, so that the debug registers cannot be read or written during the sequence. When this lock is set, accesses to some registers return errors. Only the OS Save and Restore mechanism registers can be accessed safely.

Note

An external debugger can clear this lock at any time, even if an OS Save or Restore operation is in progress.

For more information about this lock, see *OS Lock Access Register (DBGOSLAR)* on page C10-75 and *OS Lock Status Register (DBGOSLSR)* on page C10-76.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

デバッグソフトウェア イネーブル

外部デバッガは、デバッグソフトウェア イネーブル機能を使用して、デバックモニタや、システムで実行されている他のソフトウェアによってデバッグレジスタが変更されることを防止できます。デバッグソフトウェア イネーブルは、デバッグアクセス ポートの必須機能で、ARM デバッグインタフェース v5 の一部として実装されています。詳細については、『*ARM デバッグインタフェース v5、アーキテクチャ仕様書*』を参照して下さい。

DBGSWENABLE P. 付録 A-11 も参照して下さい。

注

- ソフトウェアロックと OS ロックの状態はデバッグ電力ドメインに保持され、デバッグソフトウェア イネーブルはデバッグアクセス ポートに保持されます。このため、これらのロックはコア電力ドメインの電力オフの影響を受けません。また、これらのロックはすべて、デバッグ電力ドメインのリセット時、つまり PRESETDBGn または nSYSPORESET リセットでのみ、リセット時の値に設定されます。
- SinglePower システムでは、電力オフによってソフトウェアロックと OS ロックは消失します。単一プロセッサの電力ドメインにデバッグアクセス ポートも含まれるかどうかは実装定義のため、デバッグソフトウェア イネーブルが電力オフ時に消失するかどうかも実装定義です。

C6.5.3 電力オフとアクセス許可との関係

コア電力ドメインの電力がオフになっている場合、コプロセッサインタフェース経由でのアクセスは行えません。

コア電力ドメインのレジスタへのアクセスは、そのドメインの電力がオフになっている間は行えず、アクセスを行うとエラー応答が返されます。

注

単純に書き込みを無視するのではなく、このエラー応答を返すことは、デバッガとデバッグモニタが、デバッグセッションへの割り込みを発生と同時に検出することを意味します。これにより、電力がオンになった後のセッションの再開が大幅に簡素化されます。

コア電力ドメインの電力がオフになると、デバイス電力オフおよびリセットステータス レジスタのスティッキー電力オフステータス ビット（ビット [1]）が 1 にセットされます。このビットは、コア電力ドメインの電力がオンになった後で、このレジスタの読み出しにより 0 にクリアされるまで、1 にセットされたままです。コア電力ドメインの電力がオフのままでこのレジスタが読み出されても、このビットは 1 にセットされたままです。このビットが 1 の場合、コア電力ドメインの電力がオフになっているのと同様に動作します。つまり、プロセッサはコア電力ドメイン内にあるレジスタへのアクセスを無視し、システムはエラーを返します。この動作は、レジスタが拡張 CP14 インタフェース、メモリマップされたインタフェース、外部デバッグインタフェースのいずれによりアクセスされたかにかかわらず適用されます。

この動作が有用なのは、電力オフにより内容が消失した可能性があるレジスタへ外部デバッガがアクセスを試みるとき、コア電力ドメインの電力が現在オフになっているか、電力がオンに戻ったかにかかわらず、同じ応答を得られるためです。すなわち、コア電力ドメインの電力がオフになっている期間に、外部デバッガが外部デバッグインタフェースにアクセスしなくても、プロセッサは電力オフイベントが発生したことを報告できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Debug Software Enable

An external debugger can use the Debug Software Enable function to prevent modification of the debug registers by a debug monitor or other software running on the system. The Debug Software Enable is a required function of the Debug Access Port, and is implemented as part of the ARM Debug Interface v5. For more information see the *ARM Debug Interface v5 Architecture Specification*.

See also *DBGSWENABLE* on page AppxA-11.

Note

- The states of the Software Lock and the OS Lock are held in the debug power domain, and the Debug Software Enable is in the Debug Access Port. Therefore, these locks are unaffected by the core power domain powering down. Also, all of these locks are set to their reset values only on reset of the debug power domain, that is. on a **PRESETDBGn** or **nSYSPORESET** reset.
- On SinglePower systems, the Software Lock and OS Lock are lost over a power-down. It is IMPLEMENTATION DEFINED whether the single processor power-domain also includes the Debug Access Port, and therefore also whether the Debug Software Enable is lost over a power-down.

C6.5.3 Permissions in relation to power-down

Accesses cannot be made through the coprocessor interface when the core power domain is powered down.

Access to registers in the core power domain is not possible when the domain is powered down, and accesses return an error response.

Note

Returning this error response, rather than simply ignoring writes, means that the debugger and the debug monitor detect the debug session interruption as soon as it occurs. This makes re-starting the session, after power-up, considerably easier.

When the core power domain powers down, the Sticky Power-down status bit, bit [1] of the Device Power-down and Reset Status Register, is set to 1. This bit remains set to 1 until it is cleared to 0 by a read of this register after the core power domain has powered up. If the register is read while the core power domain is still powered down, the bit remains set to 1. When this bit is 1 the behavior is as if the core power domain is powered down, meaning the processor ignores accesses to registers inside the core power domain and the system returns an error. This applies whether the register is accessed through the Extended CP14 interface, the memory-mapped interface, or the external debug interface.

This behavior is useful because when the external debugger tries to access a register whose contents might have been lost by a power-down, it gets the same response regardless of whether the core power domain is currently powered down or has powered back up. This means that, if the external debugger does not access the external debug interface during the window where the core power domain is powered down, the processor still reports the occurrence of the power-down event.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

デバッグロジックの電力がオフになっている場合、すべてのデバッグレジスタにアクセスできません。この状況では、次の規則が適用されます。

- システムは、デバッグ電力ドメインの電力がオフになっているとき、メモリマップまたは外部デバッグインタフェース経由で行われるすべてのアクセスに応答する必要があります。システムでエラー応答を生成することをお勧めします。
- コプロセッサインタフェース経由のアクセスは予測不能です。

デバッグロジックの電力がオフなのは、次の状況です。

- デバッグとコアの電力ドメインが独立している実装で、デバッグ電力ドメインの電力がオフになっているとき
- SinglePower 実装で、プロセッサの電力がオフになっているとき

C6.5.4 実装定義および予約のレジスタへのアクセス

実装定義および予約のレジスタに対するアクセスへの応答については、次に示すサブセクションを参照して下さい。

- *実装定義レジスタへのアクセス*
- *予約レジスタへのアクセス* P. C6- 29

注

ベースライン CP14 インタフェースには実装定義または予約のレジスタは存在しないため、これらのセクションの内容はベースライン CP14 インタフェース経由のアクセスには関係しません。

実装定義レジスタの空間に含まれている未使用のレジスタは、予約レジスタとして動作する必要があります。この空間は、レジスタ番号 512 ～ 575 と 928 ～ 959 です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Access to all debug registers is not possible if the debug logic is powered down. In this situation:

- the system must respond to any access made through the memory-mapped or external debug interface when the debug power domain is powered down, and ARM recommends that the system generates an error response
- accesses through the coprocessor interface are UNPREDICTABLE.

The debug logic is powered down:

- when the debug power domain is powered down, in an implementation with separate debug and core power domains
- when the processor is powered down, in a SinglePower implementation.

C6.5.4 Access to IMPLEMENTATION DEFINED and reserved registers

The following subsections describe the responses to accesses to IMPLEMENTATION DEFINED and reserved registers:

- *Access to implementation defined registers*
- *Access to reserved registers* on page C6-30.

Note

There are no IMPLEMENTATION DEFINED or reserved registers in the Baseline CP14 interface and therefore these sections do not say anything about accesses through the Baseline CP14 interface.

Any unused registers in the spaces for IMPLEMENTATION DEFINED registers must behave as reserved registers. These spaces are register numbers 512-575 and 928-959.

Access to IMPLEMENTATION DEFINED registers

When the Debug Software Enable function, described in *Permissions in relation to locks* on page C6-27, is disabling software access to the debug registers, Table C6-3 shows how the response to an accesses to an IMPLEMENTATION DEFINED register depends on the debug interface used for the access.

Table C6-3 Accesses to IMPLEMENTATION DEFINED registers when Debug Software Enable disables access

Debug interlace used for access[a]	Response
Memory-mapped interface	Error response
Extended CP14 interface	Undefined Instruction exception
External debug interface	IMPLEMENTATION DEFINED

a. There are no IMPLEMENTATION DEFINED registers in the Baseline CP14 interface.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

実装定義レジスタへのアクセス

ロックとアクセス許可との関係 P. C6-26 で説明されているデバッグソフトウェア イネーブル機能により、デバッグレジスタへのソフトウェアアクセスが不可能になっている場合、実装定義レジスタへのアクセスに対する応答は、アクセスに使用されたデバッグインタフェースによって異なります。この関係を、表 C6-3 に示します。

表 C6-3 デバッグソフトウェア イネーブルによりアクセスが不可能になっているときの実装定義レジスタへのアクセス

アクセスに使用されるデバッグインタフェース[a]	応答
メモリマップされたインタフェース	エラー応答
拡張 CP14 インタフェース	未定義命令例外
外部デバッグインタフェース	実装定義

a. ベースライン CP14 インタフェースには、実装定義レジスタは存在しません。

デバッグソフトウェア イネーブル機能によって、デバッグレジスタへのソフトウェアアクセスが不可能にされていない場合、実装定義レジスタへのアクセスに対する応答はすべて実装定義です。これは、次のいずれかが当てはまる場合、応答は実装定義であることを意味します。

- コア電力ドメインの電力がオフである。
- スティッキー電力オフ ステータスビットが 1 にセットされている。
- OS ロックが実装され、ロックされている。
- 試みられたアクセスがメモリマップされたインタフェースを使用しており、ソフトウェアロックがロックされている。

注

実装定義レジスタには、実装定義の統合レジスタであるレジスタ番号 928 ～ 959 が含まれます。

予約レジスタへのアクセス

予約レジスタへのアクセスに対する応答は、そのアクセスの試みに使用したインタフェースによって次のように異なります。

メモリマップされたインタフェース

ロックとアクセス許可との関係 P. C6-26 で説明されているデバッグソフトウェア イネーブル機能により、デバッグレジスタへのソフトウェアアクセスが不可能にされている場合、メモリマップされたインタフェース経由での予約レジスタへのアクセスはすべて、エラー応答を返します。これには、管理レジスタ空間であるレジスタ番号 832 ～ 1023 にある予約レジスタへのアクセスも含まれます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

It the Debug Software Enable function is not disabling software access to the debug registers, the response to any access to an IMPLEMENTATION DEFINED register is IMPLEMENTATION DEFINED. This means the response is IMPLEMENTATION DEFINED if any of the following apply:

- the core power domain is powered-down
- the Sticky Powered-down Status bit is set to 1
- the OS Lock is implemented and is locked
- the attempted access is using the memory-mapped interface and the Software Lock is locked.

Note

The IMPLEMENTATION DEFINED registers include the IMPLEMENTATION DEFINED integration registers, register numbers 928-959.

Access to reserved registers

The response to an access to a reserved register depends on the interface you are using to attempt the access, as follows:

Memory-mapped interface

When the Debug Software Enable function, described in *Permissions in relation to locks* on page C6-27, is disabling software access to the debug registers, any access to a reserved register through the memory-mapped interface returns an error response. This includes accesses to reserved registers in the management registers space, register numbers 832-1023.

When the Debug Software Enable function is not disabling software access to the debug registers:

- Reserved registers in the management registers space, except for reserved registers in the IMPLEMENTATION DEFINED integration registers space, are UNK/SBZP.
- For all other reserved registers, it is UNPREDICTABLE whether a register access returns an error response if any of the following applies:
  - the core power domain is powered-down
  - the Sticky Powered-Down Status bit is set to 1
  - the OS Lock is implemented and is locked
  - the Software Lock is locked.

  If none of these applies then these reserved registers are UNK/SBZP.

Extended CP14 interface

In v6 Debug and v6.1 Debug, any attempt to access a reserved register causes an Undefined Instruction exception.

In v7 Debug:

- When the Debug Software Enable function is disabling software access to the debug registers, any attempt to access a reserved register causes an Undefined Instruction exception.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

デバッグソフトウェア イネーブル機能により、デバッグレジスタへのソフトウェアアクセスが不可能にされていない場合、次の規則が適用されます。

- 管理レジスタ空間にある予約レジスタは、実装定義の統合レジスタ空間に含まれる予約レジスタを除き、UNK/SBZP です。
- 他のすべての予約レジスタについて、次のいずれかの条件が当てはまる場合、レジスタへのアクセスによってエラー応答が返されるかどうかは予測不能です。
  - コア電力ドメインの電力がオフである。
  - スティッキー電力オフ ステータスビットが 1 にセットされている。
  - OS ロックが実装され、ロックされている。
  - ソフトウェアロックがロックされている。

  これらの条件がどれも当てはまらない場合、これらの予約レジスタは UNK/SBZP です。

拡張 CP14 インタフェース

v6 デバッグと v6.1 デバッグでは、予約レジスタにアクセスする試みはすべて、未定義命令例外を引き起こします。

v7 デバッグでは、次の規則が適用されます。

- デバッグソフトウェア イネーブル機能により、デバッグレジスタへのソフトウェアアクセスが不可能にされている場合、予約レジスタにアクセスする試みはすべて、未定義命令例外を引き起こします。
- デバッグソフトウェア イネーブル機能により、デバッグレジスタへのソフトウェアアクセスが可能にされている場合、予約レジスタにアクセスする試みは次のように動作します。
  - アクセスがユーザモードで行われた場合、未定義命令例外を引き起こします。
  - アクセスが特権モードで行われた場合、予測不能です。

外部デバッグインタフェース

管理レジスタ空間にある予約レジスタは、実装定義の統合レジスタ空間に含まれる予約レジスタを除き、UNK/SBZP です。

他のすべての予約レジスタについては、次の規則が適用されます。

- 次のいずれかの条件が当てはまる場合、レジスタへのアクセスによってエラー応答が返されるかどうかは予測不能です。
  - コア電力ドメインの電力がオフである。
  - スティッキー電力オフ ステータスビットが 1 にセットされている。
  - OS ロックが実装され、ロックされている。
- これらの条件がどれも当てはまらない場合、これらの予約レジスタは UNK/SBZP です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- When the Debug Software Enable function enables software access to the debug registers, any attempt to access a reserved register:
  - — causes an Undefined Instruction exception if the access is from User mode
  - — is UNPREDICTABLE if the access is from a privileged mode.

External debug interface

Reserved registers in the management registers space, except for reserved registers in the IMPLEMENTATION DEFINED integration registers space, are UNK/SBZP.

For all other reserved registers:

- It is UNPREDICTABLE whether a register access returns an error response if any of the following applies:
  - — the core power domain is powered-down
  - — the Sticky Powered-Down Status bit is set to 1
  - — the OS Lock is implemented and is locked.
- If none of these applies then these reserved registers are UNK/SBZP.

Note

- There are no reserved registers in the Baseline CP14 interface.
- Unimplemented breakpoint and watchpoint registers are reserved registers.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

- ベースライン CP14 インタフェースには、予約レジスタは存在しません。
- 実装されていないブレークポイントおよびウォッチポイントレジスタは、予約レジスタです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.6 The CP14 debug register interfaces

This section contains the following subsections:

- *The Baseline CP14 debug register interface*
- *Extended CP14 interface* on page C6-33
- *CP14 debug registers access permissions* on page C6-36.

C6.6.1 The Baseline CP14 debug register interface

Table C6-4 lists the set of CP14 debug instructions for accessing the debug registers that must be implemented.

All MRC and MCR instructions with <coproc> = 0b1110 and <opc1> = 0b000 are debug instructions:

- Some of these instructions are defined in Table C6-4.
- Additional instructions are defined in *Extended CP14 interface* on page C6-33
- All other instructions are reserved for use by the Debug architecture. The behavior of reserved instructions is defined in *CP14 debug registers access permissions* on page C6-36.

All MRC and MCR instructions with <coproc> = 0b1110 and <opc1> = 0b001 are used by the trace extension. Other values of <opc1> are not used by the Debug architecture.

All LDC and STC instructions with <coproc> = 0b1110 that are not listed below are reserved for use by the Debug architecture and are currently UNDEFINED. All CDP, MRC2, MCR2, LDC2, STC2, LDCL, STCL, LDC2L, and STC2L instructions with <coproc> = 0b1110 are UNDEFINED.

Instructions that access registers that are only available in v7 Debug are UNDEFINED in earlier versions of the Debug architecture. For example, the read from DBGDRAR performed by MRC p14,0,<Rt>,c1,c0,0 is UNDEFINED in v6 Debug and v6.1 Debug, but is permitted in v7 Debug.

<Rt> refers to any of the general-purpose registers R0-R14. Use of APSR_nzcv is UNPREDICTABLE except where stated. Use of R13 is UNPREDICTABLE in Thumb and ThumbEE state, and is deprecated in ARM state.

Table C6-4 Baseline CP14 debug instructions

Instruction	Mnemonic	Version	Name and reference to description
MRC p14,0,<Rt>,c0,c0,0	DBGDIDR	All	*Debug ID Register (DBGDIDR)* on page C10-3
MRC p14,0,<Rt>,c1,c0,0	DBGDRAR	v7 only	*Debug ROM Address Register (DBGDRAR)* on page C10-7
MRC p14,0,<Rt>,c2,c0,0	DBGDSAR	v7 only	*Debug Self Address Offset Register (DBGDSAR)* on page C10-8

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C6.6 CP14 デバッグレジスタ インタフェース

このセクションは、次のサブセクションから構成されています。

- *ベースライン CP14 デバッグレジスタ インタフェース*
- *拡張 CP14 インタフェース* P. C6-33
- *CP14 デバッグレジスタのアクセス許可* P. C6-36

C6.6.1 ベースライン CP14 デバッグレジスタ インタフェース

デバッグレジスタにアクセスするため、必ず実装しなければならない CP14 デバッグ命令の一覧を、表 C6-4 に示します。

MRC および MCR 命令で、<coproc> = 0b1110 かつ <opc1> = 0b000 のものはすべてデバッグ命令です。

- これらの命令の一部については、表 C6-4 で定義されています。
- 追加の命令が、*拡張 CP14 インタフェース* P. C6-33 で定義されています。
- 他のすべての命令は、デバッグアーキテクチャで使用するため予約されています。予約命令の動作は、*CP14 デバッグレジスタのアクセス許可* P. C6-36 に定義されています。

MRC および MCR 命令で、<coproc> = 0b1110 かつ <opc1> = 0b001 のものはすべて、トレース拡張機能で使用されます。<opc1> の他の値は、デバッグアーキテクチャでは使用されません。

LDC および STC 命令のうち、<coproc> = 0b1110 で、以下に記載されていないものはすべて、デバッグアーキテクチャで使用するため予約されており、現在は未定義です。CDP、MRC2、MCR2、LDC2、STC2、LDCL、STCL、LDC2L、STC2L 命令で、<coproc> = 0b1110 のものはすべて未定義です。

v7 デバッグでのみ利用可能なレジスタにアクセスする命令は、以前のバージョンのデバッグアーキテクチャでは未定義です。例えば、MRC p14,0,<Rt>,c1,c0,0 による DBGDRAR からの読み出しは、v6 デバッグおよび v6.1 デバッグでは未定義ですが、v7 デバッグでは許可されます。

<Rt> は、R0 ～ R14 のうち任意の汎用レジスタを指します。APSR_nzcv の使用は、特記されていない限り予測不能です。R13 の使用は、Thumb および ThumbEE 状態では予測不能で、ARM 状態での使用は推奨されません。

表 C6-4 ベースライン CP14 のデバッグ命令

命令	ニーモニック	バージョン	名前および説明されているセクション
MRC p14,0,<Rt>,c0,c0,0	DBGDIDR	すべて	*デバッグ ID レジスタ (DBGDIDR)* P. C10-3
MRC p14,0,<Rt>,c1,c0,0	DBGDRAR	v7 のみ	*デバッグ ROM アドレスレジスタ (DBGDRAR)* P. C10-7
MRC p14,0,<Rt>,c2,c0,0	DBGDSAR	v7 のみ	*デバッグ自己アドレスオフセット レジスタ (DBGDSAR)* P. C10-8

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C6-4 Baseline CP14 debug instructions (continued)

Instruction	Mnemonic	Version	Name and reference to description
`MRC p14,0,<Rt>,c0,c5,0` `STC p14,c5,<addr_mode>`	DBGDTRRXint	All[a]	DBGDTRRX internal view. See *Host to Target Data Transfer Register (DBGDTRRX)* on page C10-40
`MCR p14,0,<Rt>,c0,c5,0` `LDC p14,c5,<addr_mode>`	DBGDTRTXint	All[a]	DBGDTRTX internal view. See *Target to Host Data Transfer Register (DBGDTRTX)* on page C10-43
`MRC p14,0,<Rt>,c0,c1,0` `MRC p14,0,APSR_nzcv,c0,c1,0`[b]	DBGDSCRint	All[a]	DBGDSCR internal view. See *Debug Status and Control Register (DBGDSCR)* on page C10-10

a. For more information, see the register description.
b. DBGDSCR[31:28] are transferred to the N, Z, C and V condition flags. For more information, see *Program Status Registers (PSRs)* on page B1-14.

C6.6.2 Extended CP14 interface

The architectural requirements for the Extended CP14 interface depend on the Debug architecture version:

v6 Debug and v6.1 Debug

All debug registers can be accessed through CP14, and implementations must provide an external access mechanism for debuggers. The details of this mechanism are not covered by the architecture specification. See *Features specific to v6 Debug and v6.1 Debug* on page C6-35.

v7 Debug The Extended CP14 interface to the debug registers is optional.

The Baseline CP14 interface is sufficient to boot-strap access to the register file, and enables software to distinguish between the Extended CP14 and memory-mapped interfaces.

See *Features specific to v7 Debug* on page C6-34.

If the Extended CP14 interface is not implemented, the memory-mapped interface must be implemented. See section *The memory-mapped and recommended external debug interfaces* on page C6-43.

Note

This section does not apply to a v7 Debug implementation that does not implement the Extended CP14 interface.

The full list of debug registers is given in Table C6-2 on page C6-18 and is not repeated here.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C6-4 ベースライン CP14 のデバッグ命令（続き）

命令	ニーモニック	バージョン	名前および説明されているセクション
`MRC p14,0,<Rt>,c0,c5,0` `STC p14,c5,<addr_mode>`	DBGDTRRXint	すべて[a]	DBGDTRRX の内部ビュー。*ホストからターゲットへのデータ転送レジスタ (DBGDTRRX)* P. C10-40 を参照して下さい。
`MCR p14,0,<Rt>,c0,c5,0` `LDC p14,c5,<addr_mode>`	DBGDTRTXint	すべて[a]	DBGDTRTX の内部ビュー。*ターゲットからホストへのデータ転送レジスタ (DBGDTRTX)* P. C10-43 を参照して下さい。
`MRC p14,0,<Rt>,c0,c1,0` `MRC p14,0,APSR_nzcv,c0,c1,0`[b]	DBGDSCRint	すべて[a]	DBGDSCR の内部ビュー。*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。

a. 詳細については、レジスタの説明を参照して下さい。
b. DBGDSCR[31:28] は、N、Z、C、V の条件フラグに転送されます。詳細については、*プログラムステータス レジスタ (PSR)* P. B1-14 を参照して下さい。

C6.6.2 拡張 CP14 インタフェース

拡張 CP14 インタフェースのアーキテクチャ上の要件は、デバッグアーキテクチャのバージョンにより次のように異なります。

v6 デバッグおよび v6.1 デバッグ

すべてのデバッグレジスタは CP14 経由でアクセス可能で、実装ではデバッガ用の外部アクセス機構を提供する必要があります。この機構の詳細は、アーキテクチャの仕様には含まれていません。*v6 デバッグと v6.1 デバッグに固有の機能* P. C6-35 を参照して下さい。

v7 デバッグ デバッグレジスタへの拡張 CP14 インタフェースはオプションです。

レジスタファイルへのブートストラップ時のアクセスにはベースライン CP14 インタフェースで十分であり、ソフトウェアは拡張 CP14 とメモリマップされたインタフェースとを区別することができます。

v7 デバッグに固有の機能 P. C6-34 を参照して下さい。

拡張 CP14 インタフェースを実装しない場合、メモリマップされたインタフェースを実装する必要があります。*メモリマップされたインタフェースと推奨される外部デバッグインタフェース* P. C6-43 を参照して下さい。

注

このセクションは、拡張 CP14 インタフェースを実装していない v7 デバッグ実装には適用されません。

デバッグレジスタの完全な一覧は P. C6-18 の表 C6-2 にあるため、ここでは記載しません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

With some exceptions, listed in *Features specific to v7 Debug* and *Features specific to v6 Debug and v6.1 Debug* on page C6-35, all the debug registers, including those in the IMPLEMENTATION DEFINED space, are accessed by the following coprocessor instructions, with `<CRn>` <= 0b0111 and the mapping shown in Figure C6-3:

- `MRC p14,0,<Rt>,<CRn>,<CRm>,<opc2>  ; Read`
- `MCR p14,0,<Rt>,<CRn>,<CRm>,<opc2>  ; Write`

Figure C6-3 Mapping from register number to CP14 instruction

For example, the instruction:

```
MRC p14,0,<Rt>,c0,c0,5
```

reads the value of DBGBCR0, that is register 80, 0b0001010000.

Features specific to v7 Debug

Table C6-5 lists the exceptions, in the v7 Debug Extended CP14 interface, to the standard mapping. In the v7 Debug Extended CP14 interface, all the instructions are UNDEFINED in User mode and UNPREDICTABLE in privileged modes.

Table C6-5 Exceptions to the standard mapping, v7 Debug with Extended CP14 interface

Register number	Name	Access	Standard mapping
33	Program Counter Sampling Register	Read-only	`MRC p14,0,<Rt>,c0,c1,2`
	Instruction Transfer Register	Write-only	`MCR p14,0,<Rt>,c0,c1,2`
40	Program Counter Sampling Register	Read-only	`MRC p14,0,<Rt>,c0,c8,2`
41	Context ID Sampling Register	Read-only	`MRC p14,0,<Rt>,c0,c9,2`
832-895	Processor identification registers	Read-only	`MRC p14,0,<Rt>,c6,c0,4` to `MRC p14,0,<Rt>,c6,c15,7`
1004	Lock Access Register	Write-only	`MCR p14,0,<Rt>,c7,c12,6`
1005	Lock Status Register	Read-only	`MRC p14,0,<Rt>,c7,c13,6`

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

*v7 デバッグに固有の機能*と *v6 デバッグと v6.1 デバッグに固有の機能* P. C6-35 に一覧表記されている一部の例外を除いて、すべてのデバッグレジスタは、実装定義空間にあるものも含めて、次に示すコプロセッサ命令に `<CRn>` <= 0b0111 を指定することでアクセスされます。マッピングを図 C6-3 に示します。

- `MRC p14,0,<Rt>,<CRn>,<CRm>,<opc2>  ; Read`
- `MCR p14,0,<Rt>,<CRn>,<CRm>,<opc2>  ; Write`

図 C6-3 レジスタ番号から CP14 命令へのマッピング

例えば、次の命令は

```
MRC p14,0,<Rt>,c0,c0,5
```

レジスタ 80(0b0001010000) である DBGBCR0 の値を読み出します。

v7 デバッグに固有の機能

v7 デバッグの拡張 CP14 インタフェースにおける標準マッピングの例外の一覧を、表 C6-5 に示します。v7 デバッグの拡張 CP14 インタフェースでは、これらの命令はすべてユーザモードでは未定義、特権モードでは予測不能です。

表 C6-5 標準マッピングの例外、拡張 CP14 インタフェース付きの v7 デバッグ

レジスタ番号	名前	アクセス	標準のマッピング
33	プログラムカウンタサンプリング レジスタ	読み出し専用	`MRC p14,0,<Rt>,c0,c1,2`
	命令転送レジスタ	書き込み専用	`MCR p14,0,<Rt>,c0,c1,2`
40	プログラムカウンタサンプリング レジスタ	読み出し専用	`MRC p14,0,<Rt>,c0,c8,2`
41	コンテキスト ID サンプリングレジスタ	読み出し専用	`MRC p14,0,<Rt>,c0,c9,2`
832-895	プロセッサ識別レジスタ	読み出し専用	`MRC p14,0,<Rt>,c6,c0,4` ～ `MRC p14,0,<Rt>,c6,c15,7`
1004	ロックアクセス レジスタ	書き込み専用	`MCR p14,0,<Rt>,c7,c12,6`
1005	ロックステータス レジスタ	読み出し専用	`MRC p14,0,<Rt>,c7,c13,6`

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accesses to the external views DBGDSCRext, DBGDTRRXext and DBGDTRTXext can be made through the standard mapping of these registers, in addition to the instructions to access the internal views DBGDSCRint, DBGDTRRXint and DBGDTRTXint provided in the Baseline CP14 interface. See *Internal and external views of the DBGDSCR and the DCC registers* on page C6-21.

Features specific to v6 Debug and v6.1 Debug

Table C6-6 lists the exceptions, in the Extended CP14 interface in v6 Debug and v6.1 Debug, to the standard mapping. All the instructions listed are UNDEFINED in ARMv6.

Table C6-6 Exceptions to the standard mapping, v6 Debug and v6.1 Debug

Register number	Name	Access	Standard mapping, all UNDEFINED
32	Host to Target Data Transfer Register	Read/write	`MRC p14,0,<Rt>,c0,c0,2`
			`MCR p14,0,<Rt>,c0,c0,2`
33	Program Counter Sampling Register	Read-only	`MRC p14,0,<Rt>,c0,c1,2`
	Instruction Transfer Register	Write-only	`MCR p14,0,<Rt>,c0,c1,2`
34	Debug Status and Control Register	Read/write	`MRC p14,0,<Rt>,c0,c2,2`
			`MCR p14,0,<Rt>,c0,c2,2`
35	Target to Host Data Transfer Register	Read/write	`MRC p14,0,<Rt>,c0,c3,2`
			`MCR p14,0,<Rt>,c0,c3,2`

See also footnote [e] on Table C6-2 on page C6-18, regarding registers 32, 33, 34, and 35.

In v6 Debug and v6.1 Debug, no debug registers map to CP14 instructions with `<CRn>` != 0b0000. All instruction encodings with `<CRn>` != 0b0000 and `<opc1>` = 0 are UNDEFINED in User mode and UNPREDICTABLE in privileged modes. All reserved encodings with `<CRn>` = 0b0000 are UNDEFINED in all modes.

Table C6-7 defines an additional ARMv6 instruction for making an internal access write to the DBGDSCR.

Table C6-7 Additional ARMv6 CP14 debug instruction

Instruction	Mnemonic	Name
`MCR p14,0,<Rt>,c0,c1,0`	DBGDSCRint	*Debug Status and Control Register (DBGDSCR)* on page C10-10

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

外部ビュー DBGDSCRext、DBGDTRRXext、DBGDTRTXext へのアクセスは、ベースライン CP14 インタフェースで提供されている内部ビュー DBGDSCRint、DBGDTRRXint、DBGDTRTXint へアクセスする命令に加えて、これらのレジスタの標準マッピング経由で行うこともできます。*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6-21 を参照して下さい。

v6 デバッグと v6.1 デバッグに固有の機能

v6 デバッグと v6.1 デバッグの拡張 CP14 インタフェースにおける、標準マッピングの例外の一覧を、表 C6-6 に示します。ここに記載されている命令はすべて、ARMv6 では未定義です。

表 C6-6 標準マッピングの例外、v6 デバッグと v6.1 デバッグ

レジスタ番号	名前	アクセス	標準のマッピング、すべて未定義
32	ホストからターゲットへのデータ転送レジスタ	読み出し / 書き込み	`MRC p14,0,<Rt>,c0,c0,2`
			`MCR p14,0,<Rt>,c0,c0,2`
33	プログラムカウンタサンプリング レジスタ	読み出し専用	`MRC p14,0,<Rt>,c0,c1,2`
	命令転送レジスタ	書き込み専用	`MCR p14,0,<Rt>,c0,c1,2`
34	デバッグステータスおよび制御レジスタ	読み出し / 書き込み	`MRC p14,0,<Rt>,c0,c2,2`
			`MCR p14,0,<Rt>,c0,c2,2`
35	ターゲットからホストへのデータ転送レジスタ	読み出し / 書き込み	`MRC p14,0,<Rt>,c0,c3,2`
			`MCR p14,0,<Rt>,c0,c3,2`

レジスタ 32、33、34、35 については、P. C6-18 の表 C6-2 の脚注 [e] も参照して下さい。

v6 デバッグと v6.1 デバッグでは、`<CRn>` != 0b0000 で CP14 命令にマッピングされるデバッグレジスタはありません。`<CRn>` != 0b0000 かつ `<opc1>` = 0 の命令エンコードはすべて、ユーザモードでは未定義、特権モードでは予測不能です。`<CRn>` = 0b0000 のすべての予約エンコードは、すべてのモードで未定義です。

DBGDSCR への内部アクセス書き込みを行うための追加 ARMv6 命令の定義を、表 C6-7 に示します。

表 C6-7 追加の ARMv6 CP14 デバッグ命令

命令	ニーモニック	名前
`MCR p14,0,<Rt>,c0,c1,0`	DBGDSCRint	*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.6.3 CP14 debug registers access permissions

By default, certain CP14 debug registers can be accessed from User mode. However, the processor can be programmed to prevent User mode access to these CP14 debug registers. For more information, see the description of the UDCCdis bit in *Debug Status and Control Register (DBGDSCR)* on page C10-10.

All CP14 debug registers can be accessed if the processor is in Debug state.

Note

When the Software Lock (DBGLAR) is implemented for a memory-mapped interface, it does not affect the behavior of CP14 instructions.

Baseline CP14 debug registers access permissions

Access to the Baseline CP14 debug registers is governed by the processor mode, Debug state and the value of DBGDSCR.UDCCdis. In addition, when the OS Lock is set accesses to the baseline registers are UNPREDICTABLE.

Note

OS Lock is implemented only in v7 Debug.

These access permissions are shown:

- in Table C6-8 for v6 Debug and v6.1 Debug
- in Table C6-9 on page C6-37 for v7 Debug

Table C6-8 Access to Baseline CP14 debug registers in v6 Debug and v6.1 Debug

Conditions			Baseline CP14 instructions[a]	DBGDSCRint writes
Debug state	Processor mode	DBGDSCR.UDCCdis[b]		
Yes	X	X	Proceed	Proceed
No	User	0	Proceed	UNDEFINED
No	User	1	UNDEFINED	UNDEFINED
No	Privileged	X	Proceed	Proceed

a. Read DBGDIDR, DBGDSCRint, DBGDTRRXint, or write DBGDTRTXint.
Attempting to use an MCR instruction to access the DBGDIDR always causes an Undefined Instruction exception.
b. DCC User mode accesses disable bit, see *Debug Status and Control Register (DBGDSCR)* on page C10-10.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C6.6.3 CP14 デバッグレジスタのアクセス許可

デフォルトでは、特定の CP14 デバッグレジスタはユーザモードからアクセスできます。ただし、これらの CP14 デバッグレジスタへのユーザモードからのアクセスを禁止するようにプロセッサをプログラム可能です。詳細については、*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 の UDCCdis ビットについての説明を参照して下さい。

プロセッサがデバッグ状態の場合、すべての CP14 デバッグレジスタにアクセス可能です。

注

メモリマップされたインタフェースにソフトウェアロック (DBGLAR) が実装されている場合、CP14 命令の動作には影響しません。

ベースライン CP14 デバッグレジスタのアクセス許可

ベースライン CP14 デバッグレジスタへのアクセスは、プロセッサのモード、デバッグ状態、および DBGDSCR.UDCCdis により決定されます。また、OS ロックが設定されている場合、ベースラインレジスタへのアクセスは予測不能です。

注

OS ロックは、v7 デバッグにのみ実装されます。

これらのアクセス許可については、次に示す表を参照して下さい。

- v6 デバッグと v6.1 デバッグについては、表 C6-8
- v7 デバッグについては、P. C6-37 の表 C6-9

表 C6-8 v6 デバッグと v6.1 デバッグでのベースライン CP14 デバッグレジスタへのアクセス

条件			ベースライン CP14 命令[a]	DBGDSCRint への書き込み
デバッグ状態	プロセッサモード	DBGDSCR.UDCCdis[b]		
はい	X	X	実行	実行
いいえ	ユーザ	0	実行	未定義
いいえ	ユーザ	1	未定義	未定義
いいえ	特権	X	実行	実行

a. DBGDIDR、DBGDSCRint、DBGDTRRXint の読み出し、または DBGDTRTXint への書き込み。
DBGDIDR へアクセスするために MCR 命令の使用を試みると、常に未定義命令例外が発生します。
b. DCC ユーザモードアクセス ディセーブルビット。*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C6-9 Access to Baseline CP14 debug registers in v7 Debug

Conditions				Baseline CP14 instructions[a]
Debug state	Processor mode	DBGDSCR.UDCCdis[b]	OS Lock	
Yes	X	X	0	Proceed
Yes	X	X	1	UNPREDICTABLE[c]
No	User	0	0	Proceed
No	User	0	1	UNPREDICTABLE[c]
No	User	1	X	UNDEFINED[d]
No	Privileged	X	0	Proceed
No	Privileged	X	1	UNPREDICTABLE[c]

a. Read DBGDIDR, DBGDSAR, DBGDRAR, DBGDSCRint, DBGDTRRXint, or write DBGDTRTXint. Attempting to use an MCR instruction to read DBGDIDR, DBGDSAR, DBGDRAR, or DBGDSCRint is UNPREDICTABLE, except in the case shown by footnote [d].
b. DCC user accesses disable bit, see *Debug Status and Control Register (DBGDSCR)* on page C10-10.
c. Apart from reads of DBGDIDR, which proceed.
d. Under these conditions, attempting to use an MCR instruction to read DBGDIDR, DBGDSAR, DBGDRAR, or DBGDSCRint always causes an Undefined Instruction exception.

Note

The Baseline CP14 instructions are not affected by:

- the recommended Debug Software Enable control in the Debug Access Port, see *Permissions in relation to locks* on page C6-27
- the Sticky Power-down status bit in the Device Power-down and Reset Status Register (DBGPRSR), see *Device Power-down and Reset Status Register (DBGPRSR), v7 Debug only* on page C10-34.

For more information on access permissions and restrictions see *Access permissions* on page C6-26.

In addition:

- if the debug power domain is powered down, instructions that access the debug registers are UNPREDICTABLE
- when the processor is in debug logic reset, reads of the debug registers return UNKNOWN values.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C6-9 v7 デバッグでのベースライン CP14 デバッグレジスタへのアクセス

条件				ベースライン CP14 命令[a]
デバッグ状態	プロセッサモード	DBGDSCR.UDCCdis[b]	OS ロック	
はい	X	X	0	実行
はい	X	X	1	予測不能[c]
いいえ	ユーザ	0	0	実行
いいえ	ユーザ	0	1	予測不能[c]
いいえ	ユーザ	1	X	未定義[d]
いいえ	特権	X	0	実行
いいえ	特権	X	1	予測不能[c]

a. DBGDIDR、DBGDSAR、DBGDRAR、DBGDSCRint、DBGDTRRXint の読み出し、または DBGDTRTXint への書き込み。
DBGDIDR、DBGDSAR、DBGDRAR、DBGDSCRint のいずれかを読み出すために MCR 命令の使用を試みると、脚注[d]に記載されている場合を除き、結果は予測不能です。
b. DCC ユーザアクセス ディセーブルビット。*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。
c. DBGDIDR の読み出しは例外で、実行されます。
d. これらの条件では、MCR 命令を使用して DBGDIDR、DBGDSAR、DBGDRAR、DBGDSCRint のいずれかの読み出しを試みると、常に未定義命令例外が引き起こされます。

注

ベースライン CP14 命令は、次の状態には影響されません。

- デバッグアクセス ポートの推奨されるデバッグソフトウェアイネーブル コントロール。*ロックとアクセス許可との関係* P. C6-26 を参照して下さい。
- デバイス電力オフおよびリセットステータス レジスタ (DBGPRSR) のスティッキー電力オフステータス ビット。*デバイス電力オフおよびリセットステータス レジスタ (DBGPRSR)、v7 デバッグのみ* P. C10-34 を参照して下さい。

アクセス許可と制限の詳細については、*アクセス許可* P. C6-25 を参照して下さい。

これらに加えて、次の規則が適用されます。

- デバッグ電力ドメインの電力がオフになっている場合、デバッグレジスタにアクセスする命令は予測不能です。
- プロセッサがデバッグロジック リセット中の場合、デバッグレジスタの読み出しにより返される値は未知です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

v7 Debug CP14 debug registers access permissions, Extended CP14 interface not implemented

Table C6-10 summarizes the complete set of CP14 instructions if the Extended CP14 interface is not implemented. In this situation, only the Baseline CP14 interface is implemented.

Table C6-10 Access to unallocated CP14 debug registers, v7 Debug with no Extended CP14 interface

Conditions		CP14 debug MCR and MRC instructions, other than Baseline CP14 instructions
Debug state	Processor mode	
Yes	X	UNPREDICTABLE
No	User	UNDEFINED
No	Privileged	UNPREDICTABLE

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

拡張 CP14 インタフェースが実装されていない場合の、v7 デバッグでの CP14 デバッグレジスタへのアクセス許可

拡張 CP14 インタフェースが実装されていない場合の CP14 命令の完全なセットの概要を、表 C6-10 に示します。この条件では、ベースライン CP14 インタフェースのみが実装されています。

表 C6-10 未割り当ての CP14 デバッグレジスタへのアクセス、v7 デバッグで拡張 CP14 インタフェースなし

条件		ベースライン CP14 命令を除く CP14 のデバッグ MCR および MRC 命令
デバッグ状態	プロセッサモード	
はい	X	予測不能
いいえ	ユーザ	未定義
いいえ	特権	予測不能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

v7 Debug CP14 debug registers access permissions, Extended CP14 interface implemented

If the Extended CP14 interface is implemented, the Debug Software Enable function can be used to prevent access to registers other than the DBGDIDR, DBGDSCR, DBGDTRRX, DBGDTRTX, DBGDSAR, DBGDRAR, DBGOSLAR, DBGOSLSR and DBGOSSRR. For more information, see *Permissions in relation to locks* on page C6-27.

For a v7 Debug implementation with the Extended CP14 interface:

- Table C6-9 on page C6-37 shows the access permissions for the Baseline CP14 debug registers
- Table C6-11 summarizes the access permissions for the other CP14 debug registers
- Table C6-12 on page C6-40 gives more information about access to the Extended CP14 interface debug registers.

Table C6-11 Access to CP14 debug registers, v7 Debug with Extended CP14 interface

Conditions[a]			Other CP14 debug instructions[b]	
Debug state	**Processor mode**	**Enable[c]**	**CRn <= 0b0111 [d]**	**CRn >= 0b1000**
Yes	X	0	UNDEFINED [e]	UNPREDICTABLE
Yes	X	1	See Table C6-12 on page C6-40[f]	UNPREDICTABLE
No	User	X	UNDEFINED	UNDEFINED
No	Privileged	0	UNDEFINED [e]	UNPREDICTABLE
No	Privileged	1	See Table C6-12 on page C6-40	UNPREDICTABLE

a. The accesses in this table are not affected by the value of the DBGDSCR.UDCCdis bit.
b. All MRC and MCR instructions with <coproc> == 0b1110 and <opc1> == 0b000 except for read accesses to DBGDIDR, DBGDSAR, DBGDRAR, DBGDSCRint, and DBGDTRRXint, and write accesses to DBGDTRTXint.
c. Debug Software Enable function is enabled.
d. Where indicated in this column, see Table C6-12 on page C6-40 for a more detailed description of access permissions to the other registers defined by the Debug architecture. In addition, there is more information about access to reserved and IMPLEMENTATION DEFINED registers in *Access to implementation defined and reserved registers* on page C6-29.
e. Except for the OS Save and Restore mechanism registers DBGOSLAR, DBGOSLSR, and DBGOSSRR, and the DBGPRSR. The state of the Debug Software Enable function does not affect access to these registers. Access to these registers must always be provided, even on implementations that do not support debug over power-down. If the implementation does not support debug over power-down the DBGOSLAR, DBGOSLSR, and DBGOSSRR are RAZWI.
f. ARM deprecates the use of these instructions from User mode in Debug state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

拡張 CP14 インタフェースが実装されている場合の、v7 デバッグでの CP14 デバッグレジスタへのアクセス許可

拡張 CP14 インタフェースが実装されている場合、デバッグソフトウェア イネーブル機能を使用して、DBGDIDR、DBGDSCR、DBGDTRRX、DBGDTRTX、DBGDSAR、DBGDRAR、DBGOSLAR、DBGOSLSR、DBGOSSRR を除くレジスタへのアクセスを禁止できます。詳細については、*ロックとアクセス許可との関係* P. C6-26 を参照して下さい。

拡張 CP14 インタフェースを持つ v7 デバッグ実装については、次に示す表を参照して下さい。

- ベースライン CP14 デバッグレジスタに対するアクセス許可を、P. C6-37 の表 C6-9 に示します。
- 他の CP14 デバッグレジスタに対するアクセス許可の概要を、表 C6-11 に示します。
- 拡張 CP14 インタフェースのデバッグレジスタに対するアクセスの詳細を、P. C6-40 の表 C6-12 に示します。

表 C6-11 CP14 デバッグレジスタへのアクセス、v7 デバッグで拡張 CP14 インタフェースあり

条件[a]			他の CP14 デバッグ命令[b]	
デバッグ状態	**プロセッサモード**	**イネーブル[c]**	**CRn <= 0b0111 [d]**	**CRn >= 0b1000**
はい	X	0	未定義 [e]	予測不能
はい	X	1	P. C6-40 の表 C6-12[f] を参照	予測不能
いいえ	ユーザ	X	未定義	未定義
いいえ	特権	0	未定義 [e]	予測不能
いいえ	特権	1	P. C6-40 の表 C6-12 を参照	予測不能

a. この表のアクセスは、DBGDSCR.UDCCdis ビットの値の影響を受けません。
b. MRC および MCR 命令で、<coproc> == b01110 かつ <opc1> == 0b000 であるもののうち、DBGDIDR、DBGDSAR、DBGDRAR、DBGDSCRint、DBGDTRRXint への読み出しアクセスと、DBGDTRTXint への書き込みアクセスを除くすべてのもの
c. デバッグソフトウェア イネーブル機能が稼動状態。
d. この列に参照先が示されている場合、デバッグアーキテクチャにより定義される他のレジスタに対するアクセス許可の詳細について、P. C6-40 の表 C6-12 を参照して下さい。また、予約および実装定義のレジスタに対するアクセスの詳細については、*実装定義および予約のレジスタへのアクセス* P. C6-28 を参照して下さい。
e. OS 保存および復元機構のレジスタ DBGOSLAR、DBGOSLSR、DBGOSSRR、DBGPRSR を除きます。デバッグソフトウェア イネーブル機能の状態は、これらのレジスタへのアクセスに影響しません。これらのレジスタへのアクセスは、実装で電力オフをまたぐデバッグをサポートしない場合でも、常に提供する必要があります。実装で電力オフをまたぐデバッグをサポートしない場合、DBGOSLAR、DBGOSLSR、DBGOSSRR は RAZ/WI です。
f. これらの命令をデバッグ状態のユーザモードから使用することは推奨されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C6-12 Access to Extended CP14 interface debug registers

Conditions		Registers:				
Sticky Power- down set	OS Lock set	DBGECR, DBGDRCR, DBGOSLAR[a], DBGOSLSR[a], DBGPRCR, DBGPRSR	DBGOSSRR[a]	Other debug[b]	All reserved[c]	Other mgmt[d]
No	No	OK	UNPREDICTABLE	OK	UNPREDICTABLE	OK
No	Yes	OK	OK	UNDEFINED	UNPREDICTABLE	OK
Yes	X	OK	UNPREDICTABLE	UNDEFINED	UNPREDICTABLE	OK

a. If the OS Save and Restore mechanism is not implemented, these registers addresses behave as reserved locations.
b. Debug register numbers 0 to 127, except for the DBGECR, DBGDRCR, the registers defined as baseline registers, and reserved registers. For details of the baseline registers see Table C6-4 on page C6-32.
c. See also *Access to implementation defined and reserved registers* on page C6-29.
d. Other management registers. This means debug register numbers 832 to 1023, except for the IMPLEMENTATION DEFINED locations, see *Access to implementation defined and reserved registers* on page C6-29.

In v7 Debug the behavior of Extended CP14 interface `MRC` and `MCR` instructions also depends on the access type of the register, as shown in Table C6-2 on page C6-18. Table C6-13 summarizes the behavior of these instructions, for:

- read accesses, using `MRC p14,0,<Rt>,<CRn>,<CRm>,<opc2>`
- write accesses, using `MCR p14,0,<Rt>,<CRn>,<CRm>,<opc2>`.

Table C6-13 Behavior of CP14 MRC and MCR instructions, v7 Debug with Extended CP14 interface

Access type[a]	Read access[b]	Write access[b]
- (Reserved)	UNPREDICTABLE	UNPREDICTABLE
Read-only	Returns register value in Rt	UNPREDICTABLE
Write-only	UNPREDICTABLE	Writes value in Rt to register
Read/write	Returns register value in Rt	Writes value in Rt to register

a. Register access type, as shown in Table C6-2 on page C6-18.
b. In a privileged mode, or in Debug state.

Some read/write registers include bits that are read-only. These bits ignore writes.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 C6- 12 拡張 CP14 インタフェースのデバッグレジスタへのアクセス

条件		レジスタ				
スティッキー電力オフがセット	OS ロックがセット	DBGECR、DBGDRCR、DBGOSLAR[a]、DBGOSLSR[a]、DBGPRCR、DBGPRSR	DBGOSSRR[a]	他のデバッグ[b]	すべての予約[c]	他の管理[d]
いいえ	いいえ	OK	予測不能	OK	予測不能	OK
いいえ	はい	OK	OK	未定義	予測不能	OK
はい	X	OK	予測不能	未定義	予測不能	OK

a. OS 保存および復元機構が実装されていない場合、これらのレジスタのアドレスは予約位置として動作します。
b. デバッグレジスタの番号 0 ～ 127、ただし DBGECR と DBGDRCR を除きます。これらのレジスタはベースラインレジスタとして定義され、予約されています。ベースラインレジスタの詳細については、P. C6- 32 の表 C6- 4 を参照して下さい。
c. *実装定義および予約のレジスタへのアクセス* P. C6- 28 も参照して下さい。
d. 他の管理レジスタ。これらは、レジスタ番号 832 ～ 1023 のうち、実装定義の位置を除いたものです。*実装定義および予約のレジスタへのアクセス* P. C6- 28 を参照して下さい。

v7 デバッグでは、拡張 CP14 インタフェースの `MRC` および `MCR` 命令の動作は、P. C6- 18 の表 C6- 2 に示されているように、レジスタのアクセスタイプにも依存します。表 C6- 13 は、次のアクセスタイプについてこれらの命令の動作を要約したものです。

- `MRC p14,0,<Rt>,<CRn>,<CRm>,<opc2>` を使用する読み出しアクセス
- `MCR p14,0,<Rt>,<CRn>,<CRm>,<opc2>` を使用する書き込みアクセス

表 C6- 13 v7 デバッグで拡張 CP14 インタフェースがある場合の、CP14 の MRC および MCR 命令の動作

アクセスタイプ[a]	読み出しアクセス[b]	書き込みアクセス[b]
–（予約）	予測不能	予測不能
読み出し専用	Rt にレジスタの値を返す	予測不能
書き込み専用	予測不能	Rt の値をレジスタに書き込む
読み出し / 書き込み	Rt にレジスタの値を返す	Rt の値をレジスタに書き込む

a. P. C6- 18 の表 C6- 2 に示されている、レジスタのアクセスタイプ
b. 特権モード、またはデバッグ状態の場合

一部の読み出し / 書き込みレジスタには、読み出し専用のビットがあります。これらのビットへの書き込みは無視されます。

プロセッサが非デバッグ状態のとき、拡張 CP14 インタフェースのレジスタに対するすべてのユーザモード アクセスは未定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When the processor is in Non-debug state, all User mode accesses to the Extended CP14 interface registers are UNDEFINED.

For example, in privileged modes the following instruction reads the value of DBGWVR7, register 103, if at least 8 watchpoints are implemented, and is UNPREDICTABLE otherwise:

```
MRC p14,0,<Rt>,c0,c7,6
```

Note

The access permissions in Table C6-11 on page C6-39 and Table C6-12 on page C6-40 have precedence over the behavior in Table C6-13 on page C6-40. For example, even if at least 8 watchpoints are implemented, the following instruction is UNDEFINED in all processor modes when the Debug Software Enable function is disabled:

```
MRC p14,0,<Rt>,c0,c7,6
```

v6 Debug and v6.1 Debug CP14 debug registers access permissions

In v6 Debug and v6.1 Debug, access to registers other than the DBGDIDR, DBGDSCR, DBGDTRRX, and DBGDTRTX is not permitted if Halting debug-mode is selected. The Debug Software Enable function, the Sticky Power-down status bit and the OS Lock are not implemented, and there are fewer CP14 debug registers than in the v7 Debug Extended CP14 interface.

For v6 Debug and v6.1 Debug:

- Table C6-8 on page C6-36 shows the access permissions for the Baseline CP14 debug registers
- Table C6-14 shows the access permissions for the other CP14 debug registers.

Table C6-14 Access to CP14 debug registers, v6 Debug and v6.1 Debug

Conditions[a]			Other CP14 debug instructions[b]
Debug state	Processor mode	DBGDSCR[15:14][c]	
Yes	X	XX	Proceed
No	User	XX	UNDEFINED
No	Privileged	00 (None)	UNDEFINED
No	Privileged	X1 (Halting)	UNDEFINED
No	Privileged	10 (Monitor)	Proceed

a. The accesses in this table are not affected by the value of the DBGDSCR.UDCCdis bit.
b. All instructions with <opc1> == 0b000 and <CRn> == 0b0000, except for read accesses to DBGDIDR, DBGDSAR, DBGDRAR, DBGDSCRint, and DBGDTRRXint, and write accesses to DBGDSCRint and DBGDTRTXint. See also Table C6-15 on page C6-42.
c. MDBGen and HDBGen bits, debug-mode enable and select bits.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

例えば、特権モードでの次の命令は、最低 8 つのウォッチポイントが実装されていれば DBGWVR7（レジスタ 103）の値を読み出し、それ以外の場合は予測不能です。

```
MRC p14,0,<Rt>,c0,c7,6
```

注

P. C6-39 の表 C6-11 と P. C6-40 の表 C6-12 に示されているアクセス許可は、P. C6-40 の表 C6-13 の動作より優先されます。例えば、最低 8 つのウォッチポイントが実装されている場合でも、デバッグソフトウェア イネーブル機能が非稼動の場合、次の命令はすべてのプロセッサモードで未定義です。

```
MRC p14,0,<Rt>,c0,c7,6
```

v6 デバッグおよび v6.1 デバッグでの CP14 デバッグレジスタへのアクセス許可

v6 デバッグおよび v6.1 デバッグでは、ホールトデバッグ モードが選択されているとき、DBGDIDR、DBGDSCR、DBGDTRRX、DBGDTRTX 以外のレジスタへのアクセスは許可されません。デバッグソフトウェア イネーブル機能、スティッキー電力オフ ステータスビット、OS ロックは実装されていません。また、CP14 デバッグレジスタの数は、v7 デバッグの拡張 CP14 インタフェースほど多くありません。

v6 デバッグおよび v6.1 デバッグについては、次の表を参照して下さい。

- ベースライン CP14 デバッグレジスタに対するアクセス許可を、P. C6-36 の表 C6-8 に示します。
- 他の CP14 デバッグレジスタに対するアクセス許可を、表 C6-14 に示します。

表 C6-14 v6 デバッグと v6.1 デバッグでの CP14 デバッグレジスタへのアクセス

条件[a]			他の CP14 デバッグ命令[b]
デバッグ状態	プロセッサモード	DBGDSCR[15:14][c]	
はい	X	XX	実行
いいえ	ユーザ	XX	未定義
いいえ	特権	00（なし）	未定義
いいえ	特権	X1（ホールト）	未定義
いいえ	特権	10（モニタ）	実行

a. この表のアクセスは、DBGDSCR.UDCCdis ビットの値の影響を受けません。
b. <opc1> == 0b000 かつ <CRn> == 0b0000 である命令のうち、DBGDIDR、DBGDSAR、DBGDRAR、DBGDSCRint、DBGDTRRXint への読み出しアクセスと、DBGDSCRint および DBGDTRTXint への書き込みアクセスを除くすべてのもの。P. C6-42 の表 C6-15 も参照して下さい。
c. MDBGen および HDBGen ビットで、デバッグモード イネーブルおよびセレクトビット。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In v6 Debug and v6.1 Debug the behavior of CP14 MRC and MCR instructions also depends on access type of the register, as shown in Table C6-2 on page C6-18. Table C6-15 summarizes the behavior, for:

- read accesses, using `MRC p14,0,<Rt>,<CRn>,<CRm>,<opc2>`
- write accesses, using `MCR p14,0,<Rt>,<CRn>,<CRm>,<opc2>`.

Table C6-15 Behavior of CP14 MRC and MCR instructions in v6 Debug and v6.1 Debug

Access type[a]	Read access	Write access
- (Reserved)	UNDEFINED	UNDEFINED
Read-only (DBGDIDR[b])	Returns register value in Rt	UNDEFINED
Write-only[c]	-	-
Read/write	Returns register value in Rt	Writes value in Rt to register

a. Register access type, as shown in Table C6-2 on page C6-18.
b. The DBGDIDR is the only read-only register in v6 Debug and v6.1 Debug.
c. There are no write-only registers in v6 Debug and v6.1 Debug.

Some read/write registers include bits that are read-only. These bits ignore writes.

For example, the following instruction reads the value of DBGWVR7, register 103, if at least 8 watchpoints are implemented, and is UNDEFINED otherwise:

```
MRC p14,0,<Rt>,c0,c7,6
```

Note

The access permissions in Table C6-14 on page C6-41 have precedence over those in Table C6-15. For example, even if at least 8 watchpoints are implemented, the following instruction is UNDEFINED in User mode, and is also UNDEFINED in privileged modes when Halting debug-mode is enabled:

```
MRC p14,0,<Rt>,c0,c7,6
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

v6 デバッグと v6.1 デバッグでは、CP14 の MRC および MCR 命令の動作は、P. C6-18 の表 C6-2 に示されているように、レジスタのアクセスタイプにも依存します。表 C6-15 は、次のアクセスタイプについてこれらの命令の動作を要約したものです。

- `MRC p14,0,<Rt>,<CRn>,<CRm>,<opc2>` を使用する読み出しアクセス
- `MCR p14,0,<Rt>,<CRn>,<CRm>,<opc2>` を使用する書き込みアクセス

表 C6-15 v6 デバッグおよび v6.1 デバッグでの CP14 MRC および MCR 命令の動作

アクセスタイプ[a]	読み出しアクセス	書き込みアクセス
–（予約）	未定義	未定義
読み出し専用 (DBGDIDR[b])	Rt にレジスタの値を返す	未定義
書き込み専用[c]	–	–
読み出し / 書き込み	Rt にレジスタの値を返す	Rt の値をレジスタに書き込む

a. P. C6-18 の表 C6-2 に示されている、レジスタのアクセスタイプ
b. v6 デバッグと v6.1 デバッグでは、DBGDIDR のみが読み出し専用レジスタです。
c. v6 デバッグと v6.1 デバッグには、書き込み専用レジスタはありません。

一部の読み出し / 書き込みレジスタには、読み出し専用のビットがあります。これらのビットへの書き込みは無視されます。

例えば、次の命令は、最低 8 つのウォッチポイントが実装されていれば DBGWVR7（レジスタ 103）の値を読み出し、それ以外の場合は未定義です。

```
MRC p14,0,<Rt>,c0,c7,6
```

注

P. C6-41 の表 C6-14 に示されているアクセス許可は、表 C6-15 のものよりも優先されます。例えば、最低 8 つのウォッチポイントが実装されていても、次の命令はユーザモードでは未定義です。また、ホールトデバッグ モードがイネーブルの場合は特権モードでも未定義です。

```
MRC p14,0,<Rt>,c0,c7,6
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.7 The memory-mapped and recommended external debug interfaces

The external debug interface is IMPLEMENTATION DEFINED in all versions of the ARM Debug architecture. This manual describes only the v7 Debug recommendations for this interface. For details of the external debug interface recommendations for v6 Debug and v6.1 Debug, contact ARM.

The memory-mapped interface to the debug registers is optional in v7 Debug.

The Baseline CP14 interface is sufficient to boot-strap access to the register file, and permits software to distinguish between the Extended CP14 and memory-mapped interfaces.

Both the memory-mapped interface and the recommended external debug interface are defined in terms of an addressable register file mapped onto a region of memory.

This section describes:

- the view of the debug registers from the processor through the memory-mapped interface
- the recommended external debug interface.

If the memory-mapped interface is not implemented, the Extended CP14 interface must be implemented, see *Extended CP14 interface* on page C6-33.

C6.7.1 Register map

The register map occupies 4KB of physical address space. The base address is IMPLEMENTATION DEFINED and must be aligned to a 4KB boundary.

Note

All memory-mapped debug registers must be mapped to Strongly-ordered or Device memory, see *Synchronization of debug register updates* on page C6-24. In systems with the ARMv7 PMSA this requirement applies even when the MPU is disabled.

Each register is mapped at an offset that is the register number multiplied by 4, the size of a word. For example, DBGWVR7, register 103, is mapped at offset `0x19C` (412).

The complete list of registers is defined in *Debug register map* on page C6-18, and is not repeated here.

C6.7.2 Shared interface port for the memory-mapped and external debug interfaces

What components in a system can access the memory-mapped interface is IMPLEMENTATION DEFINED. Typically, the processor itself and other processors in the system can access this interface. An external debugger might be able to access the debug registers through the memory-mapped interface, as well as through the external debug interface.

Because the memory-mapped interface and external debug interface share the same memory map and many of the same properties, both interfaces can be implemented as a single physical interface port to the processor.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.7 メモリマップされたインタフェースと推奨される外部デバッグインタフェース

外部デバッグインタフェースは、ARM デバッグアーキテクチャのすべてのバージョンで実装定義です。本書では、このインタフェースについては v7 デバッグでの推奨事項のみを解説します。v6 デバッグと v6.1 デバッグでの外部デバッグインタフェースに関する推奨事項の詳細については、ARM にお問い合わせ下さい。

デバッグレジスタへのメモリマップされたインタフェースは、v7 デバッグではオプションです。

レジスタファイルへのブートストラップ時のアクセスにはベースライン CP14 インタフェースで十分であり、ソフトウェアは拡張 CP14 とメモリマップされたインタフェースを区別することができます。

メモリマップされたインタフェースと、推奨される外部デバッグインタフェースはどちらも、メモリの領域にマッピングされたアドレス指定可能なレジスタファイルとして定義されています。

このセクションでは、次のトピックについて説明します。

- プロセッサからメモリマップされたインタフェース経由で見た、デバッグレジスタのビュー
- 推奨される外部デバッグインタフェース

メモリマップされたインタフェースを実装しない場合、拡張 CP14 インタフェースを実装する必要があります。*拡張 CP14 インタフェース* P. C6- 33 を参照して下さい。

C6.7.1 レジスタマップ

レジスタマップは、4KB の物理アドレス空間を占有します。ベースアドレスは実装定義ですが、4KB 境界にアラインしている必要があります。

注

メモリマップされたデバッグレジスタはすべて、ストロングリオーダまたはデバイスメモリにマッピングされる必要があります。*デバッグレジスタの更新の同期* P. C6- 23 を参照して下さい。ARMv7 PMSA を実装しているシステムでは、この条件は MPU が非稼動のときでも適用されます。

各レジスタは、レジスタ番号に 4（ワードのサイズ）をかけたオフセット位置にマッピングされます。例えば、DBGWVR7 はレジスタ番号 103 なので、オフセット `0x19C` (412) にマッピングされます。

レジスタの完全な一覧は*デバッグレジスタのマップ* P. C6- 18 に定義されているため、ここでは記載しません。

C6.7.2 メモリマップされたインタフェースおよび外部デバッグインタフェースの共有インタフェースポート

システムのどのコンポーネントが、メモリマップされたインタフェースにアクセスできるかは、実装定義です。一般に、プロセッサ自体と、システムの他のプロセッサが、このインタフェースにアクセスできます。外部デバッガは、メモリマップされたインタフェースと、外部デバッグインタフェースを経由して、デバッグレジスタにアクセスできる可能性があります。

メモリマップされたインタフェースと外部デバッグインタフェースは同じメモリマップを共有し、多くのプロパティも共有しているため、両方のインタフェースをプロセッサへの単一の物理インタフェースポートとして実装できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If the memory-mapped interface and external debug interface are implemented as a single physical interface port, external debugger accesses must be distinguishable from those of software running on a processor, including the ARM processor itself, in the target system. For example, accesses by an external debugger are not affected by the Software Lock. For the recommended memory-mapped or external debug interface this is achieved using the **PADDRDBG[31]** signal, see *PADDRDBG* on page AppxA-15.

C6.7.3 Endianness

The recommended memory-mapped and external debug interface port, referred to as the debug port, only supports word accesses. The data presented or returned on the interface is always 32 bits and is in a fixed byte order:

- bits [7:0] of the debug register are mapped to bits [7:0] of the connected data bus
- bits [15:8] of the debug register are mapped to bits [15:8] of the connected data bus
- bits [23:16] of the debug register are mapped to bits [23:16] of the connected data bus
- bits [31:24] of the debug register are mapped to bits [31:24] of the connected data bus.

The debug port ignores bits [1:0] of the address. These signals are not present in the debug port interface.

The *Debug Access Port* (DAP) and the interface between it and the debug port together form part of the external debug interface, and must support word accesses from the external debugger to these registers. The recommended ARM Debug Interface v5 (ADIv5) supports word accesses, see the *ARM Debug Interface v5 Architecture Specification* for more information. Where this interface is used the implementation must ensure that a 32-bit access by the debugger through the Debug Access Port has the same 32-bit value, in the same bit order, as the corresponding access to the debug registers. This is a requirement for tools support using ADIv5.

If a memory-mapped interface is implemented, the debug port connects to the system interconnect fabric either directly or through some form of bridge component. Such system interconnect fabrics normally support byte accesses. The system must support word-sized accesses to the debug registers. When accessing the debug registers, the behavior of an access that is smaller than word-sized is UNPREDICTABLE.

The detailed behavior of this bridge and of the system interconnect is outside the scope of the architecture.

Accesses to registers made through the debug port are not affected by the endianness configuration of the processor in which the registers reside. However, they are affected by the endianness configuration of the bus master making the access, and by the nature and configuration of the fabric that connects the two.

In an ARMv7 processor, the CPSR.E bit controls the endianness. With some assumptions, described later in this section, the operation of the CPSR.E bit is:

CPSR.E bit set to 0, for little-endian operation

If the processor reads its own DBGDIDR with an `LDR` instruction, the system ensures that the value returned in the destination register is in the same bit order as the DBGDIDR itself.

CPSR.E bit set to 1, for big-endian operation

If the processor reads its own DBGDIDR with an `LDR` instruction, the system ensures that:

- bits [7:0] of DBGDIDR are read into bits [31:24] of the destination register
- bits [15:8] of DBGDIDR are read into bits [23:16] of the destination register

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

メモリマップされたインタフェースと外部デバッグインタフェースが単一の物理インタフェースポートとして実装されている場合、外部デバッガによるアクセスを、ターゲットシステム内のプロセッサ（ARM プロセッサ自体を含む）で実行されているソフトウェアからのアクセスと区別できる必要があります。例えば、外部デバッガによるアクセスはソフトウェアロックの影響を受けません。推奨されるメモリマップまたは外部デバッグインタフェースでは、PADDRDBG[31] 信号を使用してこの動作を実現できます。*PADDRDBG* P. 付録 A-15 を参照して下さい。

C6.7.3 エンディアン形式

推奨されるメモリマップおよび外部デバッグインタフェース ポートは、デバッグポートと呼ばれ、ワードアクセスのみをサポートします。インタフェースに提示または返されるデータは常に 32 ビットで、バイト順序は固定されています。

- デバッグレジスタのビット [7:0] は、接続されているデータバスのビット [7:0] にマッピングされます。
- デバッグレジスタのビット [15:8] は、接続されているデータバスのビット [15:8] にマッピングされます。
- デバッグレジスタのビット [23:16] は、接続されているデータバスのビット [23:16] にマッピングされます。
- デバッグレジスタのビット [31:24] は、接続されているデータバスのビット [31:24] にマッピングされます。

デバッグポートは、アドレスのビット [1:0] を無視します。これらの信号は、デバッグポートインタフェースには存在しません。

デバッグアクセス ポート (DAP) とそれらの間のインタフェース、およびデバッグポートの組み合わせは、外部デバッグインタフェースの一部を構成し、外部デバッガからこれらのレジスタへのワードアクセスをサポートする必要があります。推奨される ARM デバッグインタフェース v5 (ADIv5) は、ワードアクセスをサポートします。詳細については、『*ARM デバッグインタフェース v5、アーキテクチャ仕様書*』を参照して下さい。このインタフェースを使用する場合、実装では、デバッガによりデバッグアクセス ポート経由で行われる 32 ビットアクセスが、対応するデバッグレジスタへのアクセスと同じ 32 ビット値で、ビット順序が同じであることを保証する必要があります。これは、ADIv5 の使用をサポートするツールの必要条件です。

メモリマップされたインタフェースを実装する場合、デバッグポートはシステムの相互接続構造に直接、または何かの形式のブリッジコンポーネントを経由して接続される必要があります。このようなシステムの相互接続構造は通常、バイトアクセスをサポートします。システムは、デバッグレジスタへのワードサイズのアクセスをサポートする必要があります。デバッグレジスタにアクセスするとき、ワードサイズより小さいアクセスの動作は予測不能です。

このブリッジおよびシステム相互接続の動作の詳細は、アーキテクチャの範囲外です。

デバッグポート経由で行われるレジスタへのアクセスは、そのレジスタが存在するプロセッサでのエンディアン形式の構成に影響されません。ただし、アクセスを行うバスマスタのエンディアン形式の構成、および双方を接続する構造の性質と構成の影響は受けます。

ARMv7 プロセッサでは、CPSR.E ビットがエンディアン形式を制御します。このセクションの以後で説明するいくつかの前提に基づいて、CPSR.E ビットは次のように動作します。

CPSR.E ビットが 0 にセットされ、リトルエンディアンの動作が指定されている場合

プロセッサが `LDR` 命令で自分の DBGDIDR を読み出すと、デスティネーションレジスタに返される値が DBGDIDR 自体と同じビット順序であることがシステムで保証されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- bits [23:16] of DBGDIDR are read into bits [15:8] of the destination register
- bits [31:24] of DBGDIDR are read into bits [7:0] of the destination register.

Similarly the bytes of a data value written to a debug register, for example the DBGDSCR, are reversed in big-endian configuration.

If an ARMv7 processor, with the E bit set for little-endian operation, reads the DBGDIDR of a second ARMv7 processor with an LDR instruction, then bits [7:0] of the DBGDIDR of the second processor are read into bits [7:0] of the destination register of the LDR, on the first processor. Similarly, the other bytes of the DBGDIDR are copied to the corresponding bytes of the destination register. However, if the E bit of the first processor is set for big-endian operation the bytes are reversed during the LDR operation, with bits [31:24] of the DBGDIDR of the second processor being read to bits [7:0] of the destination register of the LDR.

Note

The ordering of the bytes in the destination register on the first processor is not affected in any way by the setting of the CPSR.E bit of the second processor.

These examples assume that no additional manipulation of the data occurs in the interconnect fabric of the system. For example, an interconnect might perform byte transposition for accesses made across a boundary between a little-endian subsystem and a big-endian subsystem. Such transformations are beyond the scope of the architecture.

C6.7.4 Permission summaries for memory-mapped and external debug interfaces

This section gives summaries of the permission controls and their effects for different implementations of v7 Debug systems. The following subsections describe the access permissions for the two interfaces:

- *Access permissions for the external debug interface* on page C6-47
- *Access permissions for the memory-mapped interface* on page C6-48.

Note

For more information about access permissions in an implementation that includes the OS Save and Restore mechanism but does not provide access to the DBGOSSRR through the external debug interface, see the Note in *The OS Save and Restore mechanism* on page C6-8.

The remaining subsections apply to both interfaces:

- *Meanings of terms and abbreviations used in this section* on page C6-46
- *Permissions summary for separate debug and core power domains* on page C6-48
- *Permissions summary for SinglePower (debug and core in single power domain)* on page C6-50.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CPSR.E ビットが 1 にセットされ、ビッグエンディアンの動作が指定されている場合

プロセッサがLDR命令で自分のDBGDIDRを読み出すと、システムで次の動作が保証されます。

- DBGDIDR のビット [7:0] が、デスティネーションレジスタのビット [31:24] に読み込まれます。
- DBGDIDR のビット [15:8] が、デスティネーションレジスタのビット [23:16] に読み込まれます。
- DBGDIDR のビット [23:16] が、デスティネーションレジスタのビット [15:8] に読み込まれます。
- DBGDIDR のビット [31:24] が、デスティネーションレジスタのビット [7:0] に読み込まれます。

同様に、デバッグレジスタ、例えば DBGDSCR に書き込まれるデータの値のバイトは、ビッグエンディアンの構成では反転されます。

ARMv7 プロセッサが、E ビットがリトルエンディアン動作に設定された状態で、LDR 命令を使用して 2 番目の ARMv7 プロセッサの DBGDIDR を読み出した場合、2 番目のプロセッサの DBGDIDR のビット [7:0] は、最初のプロセッサの LDR でデスティネーションに指定されたレジスタのビット [7:0] に読み込まれます。同様に、DBGDIDR の他のバイトも、デスティネーションレジスタの対応するバイトにコピーされます。ただし、最初のプロセッサの E ビットがビッグエンディアン動作に設定されている場合は、LDR 操作でバイト順序が反転され、2 番目のプロセッサの DBGDIDR のビット [31:24] は、LDR のデスティネーションレジスタのビット [7:0] に読み込まれます。

注

最初のプロセッサのデスティネーションレジスタにおけるバイト順序は、2 番目のプロセッサの CPSR.E ビットの設定に一切の影響を受けません。

これらの例では、システムの相互接続構造で追加のデータ操作が発生しないことを想定しています。例えば、リトルエンディアンのサブシステムとビッグエンディアンのサブシステムとの境界をまたいで行われるアクセスについて、相互接続によりバイト位置の置き換えが行われることがあります。このような変換は、アーキテクチャの範囲外です。

C6.7.4 メモリマップおよび外部デバッグインタフェースのアクセス許可の概要

このセクションでは、v7 デバッグシステムの各種の実装について、アクセス許可の制御とその影響について概説します。次に示すサブセクションでは、2 つのインタフェースのアクセス許可について説明します。

- *外部デバッグインタフェースのアクセス許可* P. C6- 47
- *メモリマップされたインタフェースのアクセス許可* P. C6- 48

注

OS 保存および復元機構を含み、外部デバッグインタフェース経由の DBGOSSRR へのアクセスを提供しない実装におけるアクセス許可の詳細については、*OS 保存および復元機構* P. C6- 8 の注を参照して下さい。

以降のサブセクションは、両方のインタフェースに適用されます。

- *このセクションで使用される用語と略語の意味* P. C6- 46
- *デバッグとコアの電力ドメインが分離している場合のアクセス許可の概要* P. C6- 48
- *SinglePower（デバッグとコアが単一の電力ドメイン）でのアクセス許可の概要* P. C6- 50

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Meanings of terms and abbreviations used in this section

The following terms and abbreviations are used in the tables that summarize the access permissions:

X Don't care. The outcome does not depend on this condition.

0 The condition is false.

1 The condition is true. For more information, see Table C6-16.

IG/ABT The access is ignored or aborted.

Note

The IG/ABT response might be implemented outside the processor, for example, by the system or DAP.

Proceed The access must not be ignored, but the processor or system might return an error response. For more information about the response returned, see:

- *Permissions summary for separate debug and core power domains* on page C6-48
- *Permissions summary for SinglePower (debug and core in single power domain)* on page C6-50.

Not possible When the debug logic is powered down, accessing the debug registers is not possible. The system must respond to the access, and the response is IMPLEMENTATION DEFINED. ARM recommends that the system returns an error response.

Error Error response. Writes are ignored and reads return an UNKNOWN value.

OK Read or write access succeeds. Writes to read-only locations are ignored. Reads from RAZ or write-only locations return zero.

Some read/write registers include bits that are read-only. Unless otherwise stated in the bit description, these bits ignore writes.

UNP The access has UNPREDICTABLE results. Reads return UNKNOWN value.

DBGLAR Lock Access Register, see *Lock Access Register (DBGLAR)* on page C10-94. This is one of the management registers.

Table C6-16 lists the control conditions used in this section, and tells you where you can find more information about each of these controls. These conditions can be given an argument of X, 0 or 1, as defined at the start of this section. The table gives more information about the meaning when the argument is 1 for each condition.

Table C6-16 Meaning of (Argument = 1) for the control condition

Control condition	Meaning of (Argument = 1)	For details see
Debug logic powered	The debug power domain is powered up[a]	*Permissions in relation to power-down* on page C6-28
Core logic powered	The core power domain is powered up[a]	
Processor powered	The single power domain is powered up[a]	
Sticky power-down	DBGPRSR[1] = 1	

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

このセクションで使用される用語と略語の意味

アクセス許可の概要を示す表では、次の用語と略語が使用されます。

X 無関係。結果はこの条件の影響を受けません。

0 条件が FALSE である。

1 条件が TRUE である。詳細については、表 C6-16 を参照して下さい。

IG/ABT アクセスは無視、またはアボートされます。

注

IG/ABT 応答は、プロセッサの外側、例えばシステムや DAP に実装されることもあります。

実行 アクセスを無視できませんが、プロセッサやシステムがエラー応答を返すことはできます。返される応答の詳細については、次に示すセクションを参照して下さい。

- *デバッグとコアの電力ドメインが分離している場合のアクセス許可の概要* P. C6-48
- *SinglePower（デバッグとコアが単一の電力ドメイン）でのアクセス許可の概要* P. C6-50

不可能 デバッグロジックの電力がオフになっているとき、デバッグレジスタへのアクセスは不可能です。システムはアクセスに応答する必要があり、どのような応答を返すかは実装定義です。システムでエラー応答を返すことをお勧めします。

エラー エラー応答。書き込みは無視され、読み出しで返される値は未知です。

OK 読み出しまたは書き込みアクセスが成功します。読み出し専用位置への書き込みは無視されます。RAZ または書き込み専用位置からの読み出しは 0 を返します。

一部の読み出し / 書き込みレジスタには、読み出し専用のビットがあります。ビットの説明で特記されていない限り、これらのビットへの書き込みは無視されます。

UNP アクセスの結果は予測不能です。読み出しで返される値は未知です。

DBGLAR ロックアクセス レジスタ。*ロックアクセス レジスタ (DBGLAR)* P. C10-94 を参照して下さい。これは管理レジスタの 1 つです。

このセクションで使用される制御条件の一覧を、表 C6-16 に示します。この表には、各コントロールについての詳細情報が記載されているセクションも示されています。これらの条件には、このセクションの初めに定義されているように、X、0、1 のいずれかの引数を与えることができます。この表には、各条件の引数が 1 であるときの詳しい意味も記載されています。

表 C6-16 制御条件の（引数 = 1）の意味

制御条件	（引数 = 1）の意味	詳細の参照先
デバッグロジックの電力	デバッグ電力ドメインの電力がオンになっている[a]	*電力オフとアクセス許可との関係* P. C6-27
コアロジックの電力	コア電力ドメインの電力がオンになっている[a]	
プロセッサの電力	単一の電力ドメインの電力がオンになっている[a]	
スティッキー電力オフ	DBGPRSR[1] = 1	

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C6-16 Meaning of (Argument = 1) for the control condition (continued)

Control condition	Meaning of (Argument = 1)	For details see
OS Lock	DBGOSLSR[1] = 1	*Permissions in relation to locks* on page C6-27
Software Lock	DBGLSR[1] = 1	
Debug Software Enable	The recommended function of the DAP is enabled	

a. On a SinglePower system, the Processor powered control condition is equivalent to having both Debug logic powered and Core logic powered on a system with the recommended separate debug and core power domains.

Access permissions for the external debug interface

Table C6-17 summarizes the access permissions for the external debug interface.

When the debug logic is not powered, external debug accesses must be prohibited. An implementation can either ignore or abort these accesses.

Table C6-17 Register access permissions for the external debug interface[a]

Debug logic powered?[b]	Response	Writes or has other side-effects?
No	Not possible	-
Yes	Proceed	Yes

a. See *Meanings of terms and abbreviations used in this section* on page C6-46 when using this table.
b. Or Processor powered, on a SinglePower system.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C6-16 制御条件の（引数 = 1）の意味（続き）

制御条件	（引数 = 1）の意味	詳細の参照先
OS ロック	DBGOSLSR[1] = 1	*ロックとアクセス許可との関係* P. C6-26
ソフトウェアロック	DBGLSR[1] = 1	
デバッグソフトウェアイネーブル	DAP の推奨機能が可能である	

a. SinglePower システムでは、プロセッサの電力制御条件は、推奨されるデバッグとコアの電力ドメインが分離されているシステムでの、デバッグロジックの電力およびコアロジックの電力の両方と等価です。

外部デバッグインタフェースのアクセス許可

外部デバッグインタフェースのアクセス許可の概要を、表 C6-17 に示します。

デバッグロジックの電力がオンになっていない場合、外部デバッグアクセスを禁止する必要があります。実装では、これらのアクセスを無視またはアボートできます。

表 C6-17 外部デバッグインタフェースのレジスタアクセス許可[a]

デバッグロジックの電力[b]	応答	書き込みまたは他の副作用
いいえ	不可能	–
はい	実行	はい

a. この表を使用するときは、*このセクションで使用される用語と略語の意味* P. C6-46 を参照して下さい。
b. SinglePower システムの場合は、プロセッサの電力

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Access permissions for the memory-mapped interface

Table C6-17 on page C6-47 summarizes the access permissions for the memory-mapped interface.

At the system level, certain memory-mapped accesses must be prohibited. An implementation can either ignore or abort these accesses.

Table C6-18 Register access permissions for the memory-mapped interface[a]

Conditions:					
Debug logic powered?[b]	Debug Software Enable	Access privilege	Software Lock	Response	Writes or has other side-effects?
No	X	X	X	Not possible	-
Yes	0	X	X	IG/ABT	-
Yes	X	User	X	IG/ABT	-
Yes	1	Privileged	0	Proceed	Yes
Yes	1	Privileged	1	Proceed	DBGLAR only[c]

a. See *Meanings of terms and abbreviations used in this section* on page C6-46 when using this table.
b. Or Processor powered, on a SinglePower system.
c. Writes are ignored and reads, such as reads of DBGDSCRext, have no side-effects. Writes to the DBGLAR are permitted.

Note

If an implementation permits an external debugger to access the memory-mapped interface, it is IMPLEMENTATION DEFINED whether those accesses are controlled by the Debug Software Enable control in the debug access port.

Permissions summary for separate debug and core power domains

For implementations with separate debug and core power domains, the following tables show the effects of permissions on access to memory-mapped debug registers:

- Table C6-19 on page C6-49 for access to debug and management registers
- Table C6-20 on page C6-49 for access to the OS Save and Restore and Power-down registers.

For more information about the conditions that control access to these registers, see Table C6-16 on page C6-46.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

メモリマップされたインタフェースのアクセス許可

メモリマップされたインタフェースのアクセス許可の概要を、P. C6-47 の表 C6-17 に示します。

システムレベルでは、特定のメモリマップされたアクセスを禁止する必要があります。実装では、これらのアクセスを無視またはアボートできます。

表 C6-18 メモリマップされたインタフェースのレジスタアクセス許可[a]

条件					
デバッグロジックの電力[b]	デバッグソフトウェア イネーブル	アクセスの特権	ソフトウェアロック	応答	書き込みまたは他の副作用
いいえ	X	X	X	不可能	–
はい	0	X	X	IG/ABT	–
はい	X	ユーザ	X	IG/ABT	–
はい	1	特権	0	実行	はい
はい	1	特権	1	実行	DBGLAR のみ[c]

a. この表を使用するときは、*このセクションで使用される用語と略語の意味* P. C6-46 を参照して下さい。
b. SinglePower システムの場合は、プロセッサの電力。
c. 書き込みは無視され、DBGDSCRext などの読み出しに副作用はありません。DBGLAR への書き込みは許可されます。

注

実装で、外部デバッガがメモリマップされたインタフェースにアクセスすることが許可されている場合、それらのアクセスが、デバッグアクセス ポートのデバッグソフトウェア イネーブルコントロールによって制御されるかどうかは実装定義です。

デバッグとコアの電力ドメインが分離している場合のアクセス許可の概要

デバッグとコアの電力ドメインが分離している実装について、メモリマップされたデバッグレジスタへのアクセスに対するアクセス許可の影響は、次の表に示されています。

- デバッグおよび管理レジスタへのアクセスについては、P. C6-49 の表 C6-19 を参照して下さい。
- OS 保存および復元と電力オフレジスタへのアクセスについては、P. C6-49 の表 C6-20 を参照して下さい。

これらのレジスタへのアクセスを制御する条件の詳細については、P. C6-46 の表 C6-16 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C6-19 Debug and management register access for separate debug and core power domains[a]

Conditions			Registers:			
Core logic powered?	Sticky power-down	OS Lock	DBGDIDR, DBGECR, DBGDRCR	Other debug[b, d]	Management[c, d]	Reserved[d]
No	X	X	OK	Error	OK	UNP
Yes	0	0	OK	OK	OK	OK
Yes	0	1	OK	Error	OK	UNP
Yes	1	X	OK	Error	OK	UNP

a. See *Meanings of terms and abbreviations used in this section* on page C6-46 when using this table.
b. Registers in the memory region `0x000` - `0x1FC`, except for the DBGDIDR, DBGECR, and DBGDRCR, and reserved locations.
c. Registers in the memory region `0xD00` - `0xFFC`, except for IMPLEMENTATION DEFINED registers.
d. For details of the behavior of accesses to reserved and IMPLEMENTATION DEFINED registers see *Access to implementation defined and reserved registers* on page C6-29.

Table C6-20 OS Save and Restore and Power-down register access for separate debug and core power domains[a]

Conditions			Registers:		
Core logic powered?	Sticky power-down	OS Lock	DBGOSLSR[b] DBGPRCR, DBGPRSR	DBGOSLAR[b]	DBGOSSRR[b]
No	X	X	OK	UNP	UNP
Yes	0	0	OK	OK	UNP
Yes	0	1	OK	OK	OK
Yes	1	X	OK	OK	UNP

a. See *Meanings of terms and abbreviations used in this section* on page C6-46 when using this table.
b. If the OS Save and Restore mechanism is not implemented, these registers behave as reserved locations. For details of the behavior of accesses to reserved and IMPLEMENTATION DEFINED registers see *Access to implementation defined and reserved registers* on page C6-29.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C6-19 デバッグとコアの電力ドメインが分離している場合のデバッグおよび管理レジスタへのアクセス[a]

条件			レジスタ			
コアロジックの電力	スティッキー電力オフ	OS ロック	DBGDIDR DBGECR DBGDRCR	他のデバッグ[b, d]	管理[c、d]	予約[d]
いいえ	X	X	OK	エラー	OK	UNP
はい	0	0	OK	OK	OK	OK
はい	0	1	OK	エラー	OK	UNP
はい	1	X	OK	エラー	OK	UNP

a. この表を使用するときは、*このセクションで使用される用語と略語の意味* P. C6-46 を参照して下さい。
b. メモリ領域 `0x000` 〜 `0x1FC` のレジスタのうち、DBGDIDR、DBGECR、DBGDRCR、および予約位置を除くもの。
c. メモリ領域 `0xD00` 〜 `0xFFC` のレジスタのうち、実装定義のレジスタを除くもの。
d. 予約および実装定義のレジスタに対するアクセスの動作の詳細については、*実装定義および予約のレジスタへのアクセス* P. C6-28 を参照して下さい。

表 C6-20 デバッグとコアの電力ドメインが分離している場合の OS 保存および復元と電力オフレジスタへのアクセス[a]

条件			レジスタ		
コアロジックの電力	スティッキー電力オフ	OS ロック	DBGOSLSR[b] DBGPRCR DBGPRSR	DBGOSLAR[b]	DBGOSSRR[b]
いいえ	X	X	OK	UNP	UNP
はい	0	0	OK	OK	UNP
はい	0	1	OK	OK	OK
はい	1	X	OK	OK	UNP

a. この表を使用するときは、*このセクションで使用される用語と略語の意味* P. C6-46 を参照して下さい。
b. OS 保存および復元機構が実装されていない場合、これらのレジスタは予約位置として動作します。予約および実装定義のレジスタに対するアクセスの動作の詳細については、*実装定義および予約のレジスタへのアクセス* P. C6-28 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Permissions summary for SinglePower (debug and core in single power domain)

For implementations with a single debug and core power domain, when the processor is powered down the system response is IMPLEMENTATION DEFINED. ARM recommends that the system returns an error response, but the processor cannot generate any response. The Sticky Power-down status bit is RAZ.

Table C6-21 and Table C6-22 show the effects of permissions on access to memory-mapped debug registers.

For more information about the conditions that control access to these registers, see Table C6-16 on page C6-46.

Table C6-21 Register accesses for single debug and core power domain, part 1[a]

Conditions		Registers:		
Processor powered?	**OS Lock**	**DBGDIDR, DBGECR, DBGDRCR, DBGOSLSR[b], DBGPRCR, DBGPRSR**	**DBGOSLAR[b]**	**DBGOSSRR[b]**
No	X	Not possible	Not possible	Not possible
Yes	0	OK	OK	UNP
Yes	1	OK	OK	OK

a. See *Meanings of terms and abbreviations used in this section* on page C6-46 when using this table.
b. If the OS Save and Restore mechanism is not implemented, these registers behave as reserved locations.

Table C6-22 Register accesses for single debug and core power domain, part 2[a]

Conditions		Registers:		
Processor powered?	**OS Lock**	**Other debug[b, d]**	**Management[c, d]**	**Reserved[d]**
No	X	Not possible	Not possible	Not possible
Yes	0	OK	OK	OK
Yes	1	Error	OK	UNP

a. See *Meanings of terms and abbreviations used in this section* on page C6-46 when using this table.
b. Registers in the memory region `0x000` - `0x1FC`, except for the DBGDIDR, DBGECR, and DBGDRCR, and reserved locations.
c. Management registers, that is, registers in the memory region `0xD00` - `0xFFC`, except for IMPLEMENTATION DEFINED registers.
d. For details of the behavior of accesses to reserved and IMPLEMENTATION DEFINED registers see *Access to implementation defined and reserved registers* on page C6-29.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

SinglePower（デバッグとコアが単一の電力ドメイン）でのアクセス許可の概要

単一のデバッグおよびコア電力ドメインを持つ実装では、プロセッサの電力がオフであるときのシステム応答は実装定義です。システムがエラー応答を返すことが推奨されますが、プロセッサはどのような応答も生成できません。スティッキー電力オフステータス ビットは RAZ です。

メモリマップされたデバッグレジスタへのアクセスにおけるアクセス許可の影響を、表 C6-21 と表 C6-22 に示します。

これらのレジスタへのアクセスを制御する条件の詳細については、P. C6-46 の表 C6-16 を参照して下さい。

表 C6-21 単一のデバッグおよびコア電力ドメインでのレジスタアクセス、パート 1[a]

条件		レジスタ		
プロセッサの電力	**OS ロック**	**DBGDIDR DBGECR DBGDRCR DBGOSLSR[b] DBGPRCR DBGPRSR**	**DBGOSLAR[b]**	**DBGOSSRR[b]**
いいえ	X	不可能	不可能	不可能
はい	0	OK	OK	UNP
はい	1	OK	OK	OK

a. この表を使用するときは、*このセクションで使用される用語と略語の意味* P. C6-46 を参照して下さい。
b. OS 保存および復元機構が実装されていない場合、これらのレジスタは予約位置として動作します。

表 C6-22 単一のデバッグおよびコア電力ドメインでのレジスタアクセス、パート 2[a]

条件		レジスタ		
プロセッサの電力	**OS ロック**	**他のデバッグ[b、d]**	**管理[c、d]**	**予約[d]**
いいえ	X	不可能	不可能	不可能
はい	0	OK	OK	OK
はい	1	エラー	OK	UNP

a. この表を使用するときは、*このセクションで使用される用語と略語の意味* P. C6-46 を参照して下さい。
b. メモリ領域 `0x000` 〜 `0x1FC` のレジスタのうち、DBGDIDR、DBGECR、DBGDRCR、および予約位置を除くもの。
c. 管理レジスタ、つまりメモリ領域 `0xD00` 〜 `0xFFC` のレジスタのうち、実装定義のレジスタを除くもの。
d. 予約および実装定義のレジスタに対するアクセスの動作の詳細については、*実装定義および予約のレジスタへのアクセス* P. C6-28 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.7.5 Registers not implemented in the memory-mapped or external debug interface

In any Debug architecture version, the following registers are not implemented through the memory-mapped or external debug interfaces:

DBGDRAR *Debug ROM Address Register (DBGDRAR)* on page C10-7

DBGDSAR *Debug Self Address Offset Register (DBGDSAR)* on page C10-8.

These registers are not required by an external debugger.

In addition, there is no interface to access to DBGDSCRint, DBGDTRRXint or DBGDTRTXint through the memory-mapped or external debug interface. These operations are only available through the Baseline CP14 interface.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C6.7.5 メモリマップまたは外部デバッグインタフェースに実装されていないレジスタ

デバッグアーキテクチャのどのバージョンでも、次のレジスタはメモリマップまたは外部デバッグインタフェース経由で実装されません。

DBGDRAR *デバッグ ROM アドレスレジスタ (DBGDRAR)* P. C10-7

DBGDSAR *デバッグ自己アドレスオフセット レジスタ (DBGDSAR)* P. C10-8

これらのレジスタは、外部デバッガには必要とされません。

さらに、DBGDSCRint、DBGDTRRXint、DBGDTRTXint にメモリマップまたは外部デバッグインタフェース経由でアクセスするためのインタフェースはありません。これらの操作は、ベースライン CP14 インタフェース経由でのみ実行できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter C7
Non-invasive Debug Authentication

This chapter describes the authentication controls on non-invasive debug operations. It contains the following sections:

- *About non-invasive debug authentication* on page C7-2
- *v7 Debug non-invasive debug authentication* on page C7-4
- *Effects of non-invasive debug authentication* on page C7-6
- *ARMv6 non-invasive debug authentication* on page C7-8.

Note

The recommended external debug interface provides an authentication interface that controls both invasive debug and non-invasive debug, as described in *Authentication signals* on page AppxA-3. This chapter describes how you can use this interface to control non-invasive debug. For information about using the interface to control invasive debug see Chapter C2 *Invasive Debug Authentication*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 C7 章
非侵襲性デバッグの認証

本章では、非侵襲性デバッグ操作に対する認証制御について説明します。本章は次のセクションから構成されています。

- *非侵襲性デバッグの認証について* P. C7-2
- *v7 デバッグでの非侵襲性デバッグの認証* P. C7-4
- *非侵襲性デバッグ認証の影響* P. C7-6
- *ARMv6 での非侵襲性デバッグ認証* P. C7-8

注

推奨される外部デバッグインタフェースでは、*認証信号* P. 付録 A-3 に示されているように、侵襲性デバッグと非侵襲性デバッグの両方を制御する認証インタフェースが提供されます。本章では、このインタフェースを使用して非侵襲性デバッグを制御する方法について説明します。このインタフェースを使用して侵襲性デバッグを制御する方法については、第 C2 章*「侵襲性デバッグの認証」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C7.1 About non-invasive debug authentication

Non-invasive debug can be enabled or disabled though the external debug interface. In addition, if a processor implements the Security Extensions, non-invasive debug operations can be permitted or not permitted.

The difference between enabled and permitted is that the permitted non-invasive debug operations depend on both the security state and the operating mode of the processor. The alternatives for when non-invasive debug is permitted are:

- in all processor modes, in both Secure and Non-secure security states
- only in Non-secure state
- in Non-secure state and in Secure User mode.

Whether non-invasive debug operations are permitted in Secure User mode depends on the value of the SDER.SUNIDEN bit, see *c1, Secure Debug Enable Register (SDER)* on page B3-108.

In v6.1 Debug and v7 Debug, non-invasive debug authentication can be controlled dynamically, meaning that whether non-invasive debug is permitted can change while the processor is running, or while the processor is in Debug state. However, for more information, see *Generation of debug events* on page C3-40.

In v6 Debug, non-invasive debug authentication can be changed only while the processor is in reset.

In the recommended external debug interface, the signals that control the enabling and permitting of non-invasive debug are **DBGEN**, **SPIDEN**, **NIDEN** and **SPNIDEN**, see *Authentication signals* on page AppxA-3.

Part C of this manual assumes that the recommended external debug interface is implemented.

SPIDEN and **SPNIDEN** are only implemented on processors that implement Security Extensions. **NIDEN** is an optional signal in v6 Debug and v6.1 Debug.

Note

- **DBGEN** and **SPIDEN** also control invasive debug, see *About invasive debug authentication* on page C2-2.
- In v6 Debug and v6.1 Debug, **NIDEN** might be implemented on some non-invasive debug components and not on others. For example, the performance monitoring unit for a processor might implement **NIDEN** when the trace macrocell for the same processor does not.
- For more information about use of the authentication signals see *Changing the authentication signals* on page AppxA-4.
- For more information about ARMv6 non-invasive debug see *ARMv6 non-invasive debug authentication* on page C7-8.

If both **DBGEN** and **NIDEN** are LOW, no non-invasive debug is permitted.

Non-invasive debug authentication in v7 Debug is described in the section *v7 Debug non-invasive debug authentication* on page C7-4.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C7.1 非侵襲性デバッグの認証について

非侵襲性デバッグは、外部デバッグインタフェース経由で稼動または非稼動にできます。さらに、プロセッサにセキュリティ拡張機能が実装されている場合、非侵襲性デバッグ操作を許可または禁止することができます。

稼動と許可との相違点は、許可されている非侵襲性デバッグ操作は、セキュリティ状態とプロセッサの動作モードの両方に依存することです。非侵襲性デバッグが許可される条件は次のいずれかです。

- すべてのプロセッサモードで、セキュアおよび非セキュアセキュリティ状態の両方で
- 非セキュア状態のみで
- 非セキュア状態およびセキュアユーザ モードで

非侵襲性デバッグ操作がセキュアユーザ モードで許可されるかどうかは、SDER.SUNIDEN ビットの値に依存します。*c1、セキュアデバッグ イネーブルレジスタ (SDER)* P. B3-108 を参照して下さい。

v6.1 デバッグと v7 デバッグでは、非侵襲性デバッグの認証を動的に制御可能です。つまり、非侵襲性デバッグが許可されるかどうかを、プロセッサの実行中や、プロセッサがデバッグ状態のときに変更できます。詳細については、*デバッグ イベントの生成* P. C3-40 を参照して下さい。

v6 デバッグでは、非侵襲性デバッグの認証はプロセッサのリセット時にのみ変更可能です。

推奨される外部デバッグインタフェースでは、非侵襲性デバッグの稼動と許可を制御する信号は DBGEN、SPIDEN、NIDEN、SPNIDEN です。*認証信号* P. 付録 A-3 を参照して下さい。

本書のパート C では、推奨される外部デバッグインタフェースの実装を想定しています。

SPIDEN と SPNIDEN は、セキュリティ拡張機能を実装しているプロセッサにのみ実装されています。NIDEN は、v6 デバッグと v6.1 デバッグではオプションの信号です。

注

- DBGEN および SPIDEN は、侵襲性デバッグも制御します。*侵襲性デバッグの認証について* P. C2-2 を参照して下さい。
- v6 デバッグと v6.1 デバッグでは、NIDEN を一部の非侵襲性デバッグコンポーネントに実装し、他のコンポーネントには実装しないことが可能です。例えば、プロセッサのパフォーマンス監視ユニットに NIDEN を実装し、同じプロセッサのトレースマクロセルには実装しないことが可能です。
- 認証信号の使用法の詳細については、*認証信号の変更* P. 付録 A-4 を参照して下さい。
- ARMv6 の非侵襲性デバッグの詳細については、*ARMv6 での非侵襲性デバッグ認証* P. C7-8 を参照して下さい。

DBGEN と NIDEN の両方が LOW の場合、非侵襲性デバッグは許可されません。

v7 デバッグでの非侵襲性デバッグ認証については、*v7 デバッグでの非侵襲性デバッグの認証* P. C7-4 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The behavior of the non-invasive debug components when non-invasive debug is not enabled or not permitted is described in the following sections. These sections also describe the behavior when the processor is in Debug state:

- *Performance monitors* on page C7-6
- *Trace* on page C7-7
- *Reads of the Program Counter sampling registers* on page C8-3.

ARMv6 non-invasive debug authentication on page C7-8 describes the architectural requirements for an v6 Debug or v6.1 Debug implementation.

Note

Invasive and non-invasive debug authentication enable you to protect Secure processing from direct observation or invasion by a debugger that you do not trust. If you are designing a system you must be aware that security attacks can be aided by the invasive and non-invasive debug facilities. For example, Debug state or the DBGDSCR.INTdis bit might be used for a denial of service attack, and the Non-secure performance monitors might be used to measure the side-effects of Secure processing on Non-secure code. ARM recommends that where you are concerned about such attacks you disable invasive and non-invasive debug in all modes. However you must be aware of the limitations on the protection that debug authentication can provide, because similar attacks can be made by running malicious code on the processor in Non-secure state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

非侵襲性デバッグが稼動していない、または許可されていないときの非侵襲性デバッグコンポーネントの動作については、次に示すセクションで説明します。これらのセクションでは、プロセッサがデバッグ状態のときの動作についても説明します。

- *パフォーマンスモニタ* P. C7-6
- *トレース* P. C7-7
- *プログラムカウンタサンプリング レジスタの読み出し* P. C8-3

v6 デバッグまたは v6.1 デバッグ実装のアーキテクチャ上の要件については、*ARMv6 での非侵襲性デバッグ認証* P. C7-8 を参照して下さい。

注

侵襲性および非侵襲性デバッグの認証により、セキュアな処理が信用済みでないデバッガにより直接観測または侵襲されることを防止できます。システムを設計するときは、侵襲性および非侵襲性のデバッグ機構は、セキュリティへの攻撃に利用される恐れがあることに注意する必要があります。例えば、デバッグ状態や DBGDSCR.INTdis ビットは DoS 攻撃に使用されることがあります。また、非セキュアなパフォーマンスモニタは非セキュアなコードに対するセキュアな処理の副作用を測定するために使用されることがあります。このような攻撃の恐れがある場所では、すべてのモードで侵襲性および非侵襲性デバッグを不可能にすることをお勧めします。ただし、非セキュア状態のプロセッサで悪意のコードを実行することでも同様な攻撃が可能なため、デバッグ認証による保護には限界があることも理解しておく必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C7.2 v7 Debug non-invasive debug authentication

On processors that do not implement Security Extensions, if **NIDEN** is asserted HIGH, non-invasive debug is enabled and permitted in all modes.

If **DBGEN** is asserted HIGH the system behaves as if **NIDEN** is asserted HIGH, regardless of the actual state of the **NIDEN** signal.

Table C7-1 shows the required behavior in v7 Debug when the Security Extensions are not implemented.

Table C7-1 v7 Debug non-invasive debug authentication, Security Extensions not implemented

DBGEN	NIDEN	Modes in which non-invasive debug is permitted
LOW	LOW	None. Non-invasive debug is disabled.
x	HIGH	All modes.
HIGH	LOW	All modes.

On a processor that implements the Security Extensions:

- If both **NIDEN** and **SPNIDEN** are asserted HIGH, non-invasive debug is enabled and permitted in all modes and security states.
- If **NIDEN** is HIGH and **SPNIDEN** is LOW:
  - — non-invasive debug is enabled and permitted in Non-secure state
  - — non-invasive debug is not permitted in Secure privileged modes
  - — whether non-invasive debug is permitted in Secure User mode depends on the value of the SDER.SUNIDEN bit.

If **DBGEN** is HIGH, the system behaves as if **NIDEN** is HIGH, regardless of the actual state of the **NIDEN** signal

If **SPIDEN** is HIGH, the system behaves as if **SPNIDEN** is HIGH, regardless of the actual state of the **SPNIDEN** signal.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C7.2 v7 デバッグでの非侵襲性デバッグの認証

セキュリティ拡張機能を実装していないプロセッサでは、NIDEN が HIGH にアサートされている場合、非侵襲性デバッグが可能で、すべてのモードで許可されています。

DBGEN が HIGH にアサートされている場合、システムは NIDEN 信号の実際の状態にかかわらず、NIDEN が HIGH にアサートされているのと同様に動作します。

セキュリティ拡張機能が実装されていない場合に v7 デバッグで要求される動作を、表 C7-1 に示します。

表 C7-1 セキュリティ拡張機能が実装されていない場合の v7 デバッグ非侵襲性デバッグ認証

DBGEN	NIDEN	非侵襲性デバッグが許可されるモード
LOW	LOW	なし。非侵襲性デバッグは不可能です。
x	HIGH	すべてのモード
HIGH	LOW	すべてのモード

セキュリティ拡張機能を実装しているプロセッサでは、次の動作が行われます。

- NIDEN と SPNIDEN の両方が HIGH にアサートされている場合、非侵襲性デバッグは可能で、すべてのモードとセキュリティ状態で許可されています。
- NIDEN が HIGH で SPNIDEN が LOW の場合、次の動作が行われます。
  - – 非侵襲性デバッグは可能で、非セキュア状態で許可されています。
  - – セキュア特権モードでは、非侵襲性デバッグは許可されていません。
  - – 非侵襲性デバッグがセキュアユーザ モードで許可されるかどうかは、SDER.SUNIDEN ビットの値に依存します。

DBGEN が HIGH の場合、システムは NIDEN 信号の実際の状態にかかわらず、NIDEN が HIGH であるのと同様に動作します。

SPIDEN が HIGH の場合、システムは SPNIDEN 信号の実際の状態にかかわらず、SPNIDEN が HIGH であるのと同様に動作します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C7-2 shows the non-invasive debug authentication for ARMv7 processors that implement the Security Extensions.

Table C7-2 v7 Debug non-invasive debug authentication, Security Extensions implemented

Signals				SDER. SUNIDEN	Modes in which non-invasive debug is permitted
DBGEN	NIDEN	SPIDEN	SPNIDEN		
LOW	LOW	x	x	x	None. Non-invasive debug is disabled.
LOW	HIGH	LOW	LOW	0	All modes in Non-secure state
LOW	HIGH	LOW	LOW	1	All modes in Non-secure state, Secure User mode.
LOW	HIGH	LOW	HIGH	x	All modes in both security states.
LOW	HIGH	HIGH	x	x	All modes in both security states.
HIGH	x	LOW	LOW	0	All modes in Non-secure state.
HIGH	x	LOW	LOW	1	All modes in Non-secure state, Secure User mode.
HIGH	x	LOW	HIGH	x	All modes in both security states.
HIGH	x	HIGH	x	x	All modes in both security states.

Note

The value of the SDER.SUIDEN bit does not have any effect on non-invasive debug.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

セキュリティ拡張機能を実装している ARMv7 プロセッサでの非侵襲性デバッグ認証を、表 C7-2 に示します。

表 C7-2 セキュリティ拡張機能が実装されている場合の v7 デバッグ非侵襲性デバッグ認証

信号				SDER SUNIDEN	非侵襲性デバッグが許可されるモード
DBGEN	NIDEN	SPIDEN	SPNIDEN		
LOW	LOW	x	x	x	なし。非侵襲性デバッグは不可能です。
LOW	HIGH	LOW	LOW	0	非セキュア状態のすべてのモード
LOW	HIGH	LOW	LOW	1	非セキュア状態のすべてのモード、セキュアユーザ モード
LOW	HIGH	LOW	HIGH	x	両方のセキュリティ状態のすべてのモード
LOW	HIGH	HIGH	x	x	両方のセキュリティ状態のすべてのモード
HIGH	x	LOW	LOW	0	非セキュア状態のすべてのモード
HIGH	x	LOW	LOW	1	非セキュア状態のすべてのモード、セキュアユーザ モード
HIGH	x	LOW	HIGH	x	両方のセキュリティ状態のすべてのモード
HIGH	x	HIGH	x	x	両方のセキュリティ状態のすべてのモード

注

SDER.SUIDEN ビットの値は、非侵襲性デバッグには一切の影響を及ぼしません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C7.3 Effects of non-invasive debug authentication

The following sections describe the effects of the non-invasive debug authentication on the non-invasive debug components:

- *Performance monitors*
- *Trace* on page C7-7
- *Reads of the Program Counter sampling registers* on page C8-3.

C7.3.1 Performance monitors

Performance monitors provide a non-invasive debug feature, and are controlled by the non-invasive debug authentication signals. For more information, see Chapter C9 *Performance Monitors*.

The cycle counter, PMCCNTR, is not controlled by the non-invasive debug authentication signals. However, setting the PMCR.DP flag to 1 disables PMCCNTR counting in regions of code where the event counters are disabled. For details see *c9, Performance Monitor Control Register (PMCR)* on page C10-105.

Table C7-3 describes the behavior of the performance monitors when non-invasive debug is disabled or not permitted, and in Debug state.

Table C7-3 Behavior of performance monitors when non-invasive debug not permitted

Debug state	Non-invasive debug permitted and enabled	PMCR.DP[a]	Event counters enabled and events exported[a, b]	PMCCNTR enabled
Yes	x	x	No	No
No	Yes	x	Yes	Yes
No	No	0	No	Yes
No	No	1	No	No

a. See *c9, Performance Monitor Control Register (PMCR)* on page C10-105.
b. The events are exported only if the PMCR.X bit is set to 1.

The performance monitors are not intended to be completely accurate, see *Accuracy of the performance monitors* on page C9-5. In particular, some inaccuracy is permitted at the point of changing security state. However, to avoid the leaking of information from the Secure state, the permitted inaccuracy is that non-prohibited transactions can be uncounted. Prohibited transactions must not be counted.

Entry to and exit from Debug state can also disturb the normal running of the processor, causing additional inaccuracy in the performance monitors. Disabling the counters while in Debug state limits the extent of this inaccuracy. Implementations can limit this inaccuracy to a greater extent, for example by disabling the counters as soon as possible during the Debug state entry sequence.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C7.3 非侵襲性デバッグ認証の影響

次に示すセクションでは、非侵襲性デバッグ認証が非侵襲性デバッグコンポーネントに及ぼす影響について説明します。

- *パフォーマンスモニタ*
- *トレース* P. C7-7
- *プログラムカウンタサンプリング レジスタの読み出し* P. C8-3

C7.3.1 パフォーマンスモニタ

パフォーマンスモニタは非侵襲性デバッグ機能を提供し、非侵襲性デバッグ認証信号によって制御されます。詳細については、第 C9 章*「パフォーマンスモニタ」*を参照して下さい。

サイクルカウンタ (PMCCNTR) は、非侵襲性デバッグ認証信号によって制御されません。ただし、PMCR.DP フラグを 1 にセットすると、イベントカウンタが非稼動であるコード領域について、PMCCNTR のカウントが非稼動になります。詳細については、*c9、パフォーマンスモニタ制御レジスタ (PMCR)* P. C10-105 を参照して下さい。

非侵襲性デバッグが非稼動または許可されておらず、デバッグ状態であるときのパフォーマンスモニタの動作を、表 C7-3 に示します。

表 C7-3 非侵襲性デバッグが許可されていないときのパフォーマンスモニタの動作

デバッグ状態	非侵襲性デバッグが許可され稼動中である	PMCR.DP[a]	イベントカウンタが稼動中でイベントが出力されている[a,b]	PMCCNTR が稼動中
はい	x	x	いいえ	いいえ
いいえ	はい	x	はい	はい
いいえ	いいえ	0	いいえ	はい
いいえ	いいえ	1	いいえ	いいえ

a. 詳細については、*c9、パフォーマンスモニタ制御レジスタ (PMCR)* P. C10-105 を参照して下さい。
b. イベントは、PMCR.X ビットが 1 にセットされている場合のみ出力されます。

パフォーマンスモニタは、完全に正確なことを意図したものではありません。*パフォーマンスモニタの正確性* P. C9-5 を参照して下さい。特に、セキュリティ状態の変更時には多少の不正確さが許容されます。ただし、セキュリティ状態から情報が漏洩することを防止するため、許容される不正確さは、禁止されていないトランザクションがカウントされなくてもよいというものです。禁止されているトランザクションはカウントされてはいけません。

デバッグ状態の開始および終了によっても、プロセッサの通常の実行が中断され、パフォーマンスモニタがさらに不正確になることがあります。デバッグ状態の間にカウンタを非稼動にすると、この不正確さの程度を制限できます。実装では、デバッグ状態開始シーケンスのできるだけ早い段階でカウンタを非稼動にするなどの方法を使用して、この不正確さをより抑えることが可能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C7.3.2 Trace

All instructions and data transfers are ignored by the trace device when:

- non-invasive debug is disabled
- the processor is in a mode or state where non-invasive debug is not permitted
- the processor is in Debug state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C7.3.2 トレース

次の条件では、すべての命令とデータ転送はトレースデバイスによって無視されます。

- 非侵襲性デバッグが非稼動である。
- プロセッサが、非侵襲性デバッグが許可されないモードまたは状態である。
- プロセッサがデバッグ状態である。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C7.4 ARMv6 non-invasive debug authentication

An ARMv6 processor might implement the v7 Debug non-invasive debug authentication signaling described in *v7 Debug non-invasive debug authentication* on page C7-4.

In general, non-invasive debug authentication in ARMv6 Debug is IMPLEMENTATION DEFINED. For details of the implemented authentication scheme you must see the appropriate product documentation. In particular:

- it is IMPLEMENTATION DEFINED whether the **NIDEN** signal is implemented
- the exact roles of the following signals are IMPLEMENTATION DEFINED:
  - **DBGEN**, **SPIDEN**, and **SPNIDEN**
  - **NIDEN**, if it is implemented.

However, an ARMv6 non-invasive debug authentication scheme must obey the following rules:

- If **NIDEN** is implemented then tying **NIDEN** and **DBGEN** both LOW guarantees that non-invasive debug is disabled.
- if **NIDEN** is not implemented then the mechanism for disabling non-invasive debug is IMPLEMENTATION DEFINED. An implementation might not support any mechanism for disabling non-invasive debug.
- When the Security Extensions are implemented, tying **SPIDEN** and **SPNIDEN** both LOW guarantees that non-invasive debug is not permitted in Secure privileged modes.

  In addition, if **SPIDEN** and **SPNIDEN** are both LOW then setting SDER.SUNIDEN to 0 guarantees that non-invasive debug is not permitted in Secure User mode.

  If non-invasive debug is enabled then if SDER.SUNIDEN is 1, non-invasive debug is permitted in Secure User mode.
- If **NIDEN** is implemented then tying **NIDEN** and **SPNIDEN** both HIGH is guaranteed to enable and permit non-invasive debug in all modes in both security states.

  If **NIDEN** is not implemented then tying **SPNIDEN** HIGH is guaranteed to enable and permit non-invasive debug in all modes in both security states.

Table C7-4 shows the architectural requirements for non-invasive debug behavior in an ARMv6 Debug implementation that does not include the Security Extensions.

Table C7-4 ARMv6 non-invasive debug authentication requirements, Security Extensions not implemented

NIDEN	DBGEN	Non-invasive debug behavior
Implemented and LOW	LOW	Disabled.
Implemented and HIGH	x	Enabled.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C7.4 ARMv6 での非侵襲性デバッグ認証

ARMv6 プロセッサは、*v7 デバッグでの非侵襲性デバッグの認証* P. C7-4 に説明されている v7 デバッグの非侵襲性デバッグ認証信号処理を実装することもできます。

一般に、ARMv6 デバッグでの非侵襲性デバッグ認証は実装定義です。実装されている認証手法の詳細については、該当する製品説明書を参照する必要があります。特に、次の項目について注意が必要です。

- NIDEN 信号が実装されるかどうかは実装定義です。
- 次の信号の正確な役割は実装定義です。
  - DBGEN、SPIDEN、SPNIDEN
  - NIDEN（実装されている場合）

ただし、ARMv6 の非侵襲性デバッグ認証方式は、次の規則に従う必要があります。

- NIDEN が実装されている場合、NIDEN と DBGEN の両方を LOW に直結することで、非侵襲性デバッグが不可能になることが保証されます。
- NIDEN が実装されていない場合、非侵襲性デバッグを不可能にする機構は実装定義です。実装で、非侵襲性デバッグを不可能にする機構がサポートされていない可能性もあります。
- セキュリティ拡張機能が実装されている場合、SPIDEN と SPNIDEN の両方を LOW に直結することで、非侵襲性デバッグがセキュア特権モードで許可されないことが保証されます。

  さらに、SPIDEN と SPNIDEN の両方が LOW の場合、SDER.SUNIDEN を 0 にセットすることで、非侵襲性デバッグがセキュアユーザ モードで許可されないことが保証されます。

  非侵襲性デバッグが可能な場合、SDER.SUNIDEN が 1 であれば、非侵襲性デバッグがセキュアユーザ モードで許可されます。
- NIDEN が実装されている場合、NIDEN と SPNIDEN の両方を HIGH に直結することで、非侵襲性デバッグが可能になり、両方のセキュリティ状態について、すべてのモードで許可されることが保証されます。

  NIDEN が実装されていない場合、SPNIDEN を HIGH に直結することで、非侵襲性デバッグが可能になり、両方のセキュリティ状態について、すべてのモードで許可されることが保証されます。

セキュリティ拡張機能を含まない ARMv6 デバッグ実装での、非侵襲性デバッグの動作に関するアーキテクチャ上の必要条件を、表 C7-4 に示します。

表 C7-4 セキュリティ拡張機能が実装されていない場合の ARMv6 非侵襲性デバッグ認証の必要条件

NIDEN	DBGEN	非侵襲性デバッグの動作
実装され LOW	LOW	不可能
実装され HIGH	x	可能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C7-5 shows the architectural requirements for non-invasive debug behavior in an ARMv6 Debug implementation that includes the Security Extensions.

Table C7-5 ARMv6 non-invasive debug authentication requirements, Security Extensions implemented

Signals					Non-invasive debug behavior
NIDEN	DBGEN	SPIDEN	SPNIDEN	SDER. SUNIDEN	
Implemented and LOW	LOW	x	x	x	Disabled.
x	x	LOW	LOW	0	Not permitted in all modes in Secure state.
x	x	LOW	LOW	1	Not permitted in Secure privileged modes. Permitted in Secure User mode if enabled.
Implemented and HIGH	x	x	x	x	Permitted in all modes in Non-secure state. Might also be permitted in Secure state.
Implemented and HIGH	x	x	HIGH	x	Permitted in all modes and security states.
Not implemented	x	x	HIGH	x	Permitted in all modes and security states.

An ARMv6 Debug implementation that includes the Security Extensions might have other signal combinations that permit non-invasive debug in Secure privileged modes. You must take care to avoid unknowingly permitting non-invasive debug.

There is no mechanism that a debugger can use to determine the implemented mechanism for controlling non-invasive debug on an ARMv6 processor. You must see the product documentation for this information.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

セキュリティ拡張機能を含む ARMv6 デバッグ実装での、非侵襲性デバッグの動作に関するアーキテクチャ上の必要条件を、表 C7-5 に示します。

表 C7-5 セキュリティ拡張機能が実装されている場合の ARMv6 非侵襲性デバッグ認証の必要条件

信号					非侵襲性デバッグの動作
NIDEN	DBGEN	SPIDEN	SPNIDEN	SDER SUNIDEN	
実装され LOW	LOW	x	x	x	不可能
x	x	LOW	LOW	0	セキュア状態のすべてのモードで禁止
x	x	LOW	LOW	1	セキュア特権モードで禁止。可能な場合、セキュアユーザ モードで許可
実装され HIGH	x	x	x	x	非セキュア状態のすべてのモードで許可。セキュア状態でも許可される可能性があります。
実装され HIGH	x	x	HIGH	x	すべてのモードとセキュリティ状態で許可
実装なし	x	x	HIGH	x	すべてのモードとセキュリティ状態で許可

セキュリティ拡張機能を含む ARMv6 デバッグ実装では、他の信号の組み合わせにより、セキュア特権モードで非侵襲性デバッグが許可されることがあります。気付かずに非侵襲性デバッグを許可しないよう注意して下さい。

ARMv6 プロセッサで、非侵襲性デバッグの制御のために実装されている機構を特定するため、デバッガが使用できる機構は存在しません。この情報については、製品説明書を参照する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter C8
Sample-based Profiling

This chapter describes sample-based profiling. Sample-based profiling is an optional non-invasive debug component. It contains the following section:

- *Program Counter sampling* on page C8-2.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 C8 章
サンプルベースのプロファイリング

本章では、サンプルベースのプロファイリングについて説明します。サンプルベースのプロファイリングは、オプションの非侵襲性デバッグコンポーネントです。本章は次のセクションから構成されています。

- *プログラムカウンタのサンプリング* P. C8-2

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C8.1 Program Counter sampling

In ARMv6, the *Program Counter Sampling Register* (DBGPCSR) is an optional part of the recommended external debug interface. It is not defined by the architecture.

In v7 Debug, Program Counter sampling is an optional feature defined by the architecture. The following sections describe this feature:

- *Implemented Program Counter sampling registers*
- *Reads of the Program Counter sampling registers* on page C8-3

C8.1.1 Implemented Program Counter sampling registers

In v7 Debug, it is IMPLEMENTATION DEFINED whether the DBGPCSR is implemented. It is an optional extension to the Debug architecture, that provides a mechanism for coarse-grained profiling of code executing on the processor without changing the behavior of that code. For details see *Program Counter Sampling Register (DBGPCSR)* on page C10-38.

If the DBGPCSR is implemented, it is IMPLEMENTATION DEFINED whether a second sampling register is also implemented. This register is the Context ID Sampling Register (DBGCIDSR) and is described in *Context ID Sampling Register (DBGCIDSR)* on page C10-39.

If a processor does not implement DBGPCSR it does not implement DBGCIDSR.

If a processor implements only DBGPCSR, it is IMPLEMENTATION DEFINED whether it is implemented as register 33, as register 40, or as both register 33 and register 40.

If a processor implements both DBGPCSR and DBGCIDSR:

- it must implement:
  - DBGPCSR as register 40
  - DBGCIDSR as register 41
- it is IMPLEMENTATION DEFINED whether it also implements DBGPCSR as register 33.

If a processor implements DBGPCSR as both register 33 and register 40, the two register numbers are aliases of a single register. ARM deprecates reading DBGPCSR as register 33 on an implementation that also implements it as register 40.

To determine which, if any, of the Program Counter sampling registers are implemented, and the register numbers used for any implemented registers, read:

- the DEVID_imp and PCSR_imp bits of the DBGDIDR, see *Debug ID Register (DBGDIDR)* on page C10-3
- the DBGDEVID.PCsample field, see *Debug Device ID Register (DBGDEVID)* on page C10-6.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C8.1 プログラムカウンタのサンプリング

ARMv6 では、*プログラムカウンタサンプリング レジスタ*(DBGPCSR) は、推奨される外部デバッグインタフェースのオプション部分です。アーキテクチャでは定義されていません。

v7 デバッグでは、プログラムカウンタ サンプリングはアーキテクチャで定義されているオプション機能です。この機能の説明については、次に示すセクションを参照して下さい。

- *実装されているプログラムカウンタサンプリング レジスタ*
- *プログラムカウンタサンプリング レジスタの読み出し* P. C8-3

C8.1.1 実装されているプログラムカウンタサンプリング レジスタ

v7 デバッグでは、DBGPCSR を実装するかどうかは実装定義です。これはデバッグアーキテクチャのオプションの拡張機能で、プロセッサで実行中のコードの動作を変更することなく、粒度の粗いプロファイリングを行う機構を提供します。詳細については、*プログラムカウンタサンプリング レジスタ (DBGPCSR)* P. C10-38 を参照して下さい。

DBGPCSR が実装されている場合、2 番目のサンプリングレジスタも実装するかどうかは実装定義です。このレジスタはコンテキスト ID サンプリングレジスタ (DBGCIDSR) で、*コンテキスト ID サンプリングレジスタ (DBGCIDSR)* P. C10-39 に説明されています。

プロセッサに DBGPCSR が実装されていない場合、DBGCIDSR も実装されていません。

プロセッサに DBGPCSR のみが実装されている場合、レジスタ 33 として実装されているか、レジスタ 40 として実装されているか、レジスタ 33 と 40 の両方として実装されているかは実装定義です。

プロセッサに DBGPCSR と DBGCIDSR の両方が実装されている場合、次の規則が適用されます。

- 次のレジスタも実装する必要があります。
  - DBGPCSR をレジスタ 40 として
  - DBGCIDSR をレジスタ 41 として
- DBGPCSR をレジスタ 33 としても実装するかどうかは実装定義です。

プロセッサに DBGPCSR がレジスタ 33 およびレジスタ 40 の両方として実装されている場合、これら 2 つのレジスタ番号は単一のレジスタのエイリアスです。DBGPCSR をレジスタ 40 としても実装している実装で、レジスタ 33 として読み出すことは推奨されません。

プログラムカウンタサンプリング レジスタが実装されているか、実装されている場合にはどれが実装されているか、および実装されているレジスタに使用されているレジスタ番号を判定するには、次のレジスタを読み出します。

- DBGDIDR の DEVID_imp および PCSR_imp ビット。*デバッグ ID レジスタ (DBGDIDR)* P. C10-3 を参照して下さい。
- DBGDEVID.PCsample フィールド。*デバッグデバイス ID レジスタ (DBGDEVID)* P. C10-6 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

ARM recommends that an implementation that supports sample-based profiling:

- implements both DBGPCSR and DBGCIDSR
- implements DBGPCSR as register 40
- also implements DBGPCSR as register 33, for backwards compatibility with implementations that implement it only as register 33.

C8.1.2 Reads of the Program Counter sampling registers

A read of the DBGPCSR:

- Normally:
  - — returns the address of an instruction *recently executed* by the processor
  - — sets the DBGCIDSR, if implemented, to the current value of the CONTEXTIDR.

  For more information about the CONTEXTIDR, see:
  - — *c13, Context ID Register (CONTEXTIDR)* on page B3-153, for a VMSA implementation
  - — *c13, Context ID Register (CONTEXTIDR)* on page B4-76, for a PMSA implementation.
- Alternatively, when any of the following is true, returns `0xFFFFFFFF` and sets the DBGCIDSR, if implemented, to an UNKNOWN value:
  - — non-invasive debug is disabled
  - — the processor is in a mode or state where non-invasive debug is not permitted
  - — the processor is in Debug state.

If the DBGCIDSR is implemented, reading it returns the last value to which it was set.

Note

The ARM architecture does not define *recently executed*. The delay between an instruction being executed by the processor and its address appearing in the DBGPCSR is not defined. For example, if a piece of code reads the DBGPCSR of the processor it is running on, there is no guaranteed relationship between the program counter for that piece of code and the value read. The DBGPCSR is intended only for use by an external agent to provide statistical information for code profiling.

The value in the DBGPCSR always references a committed instruction. An implementation must not sample values that reference instructions that are fetched but not committed for execution.

If DBGPCSR is implemented, it must be possible to sample references to branch targets. It is IMPLEMENTATION DEFINED whether references to other instructions can be sampled. ARM recommends that a reference to any instruction can be sampled.

The branch target for a conditional branch instruction that fails its condition code check is the instruction that follows the conditional branch instruction. The branch target for an exception is the exception vector address.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

サンプルベースのプロファイリングをサポートする実装では、次の実装をお勧めします。

- DBGPCSR と DBGCIDSR の両方を実装する。
- DBGPCSR をレジスタ 40 として実装する。
- DBGPCSR をレジスタ 33 としても実装する。これは、レジスタ 33 としてのみこのレジスタを実装している実装との下位互換性のためです。

C8.1.2 プログラムカウンタサンプリング レジスタの読み出し

DBGPCSR の読み出しは、次のように行われます。

- 通常
  - – プロセッサによって*最近実行された*命令のアドレスを返します。
  - – DBGCIDSR が実装されている場合、CONTEXTIDR の現在の値に設定されます。

  CONTEXTIDR の詳細については、次の説明を参照して下さい。
  - – *c13、コンテキスト ID レジスタ (CONTEXTIDR)* P. B3-153（VMSA 実装の場合）
  - – *c13、コンテキスト ID レジスタ (CONTEXTIDR)* P. B4-76（PMSA 実装の場合）
- または、次のいずれかの条件が真の場合は `0xFFFFFFFF` を返し、DBGCIDSR が実装されている場合は未知の値に設定されます。
  - – 非侵襲性デバッグが非稼動である。
  - – プロセッサが、非侵襲性デバッグが許可されないモードまたは状態である。
  - – プロセッサがデバッグ状態である。

DBGCIDSR が実装されている場合、読み出しを行うと、最後に設定された値が返されます。

注

ARM アーキテクチャでは、*最近実行された*が定義されていません。プロセッサによって実行されている命令と、DBGPCSR に現れるアドレスとの間の遅延は定義されていません。例えば、コードの一部が実行中に、自分が実行されているプロセッサの DBGPCSR を読み出した場合、コードのその部分に対応するプログラムカウンタと、読み出される値との関係は保証されていません。DBGPCSR は、外部エージェントがコードプロファイリングのための統計情報を提供するために使用することのみを意図しています。

DBGPCSR の値は、必ずコミットされた命令を参照しています。実装では、フェッチされたが実行にコミットされていない命令を参照している値をサンプリングすることはできません。

DBGPCSR が実装されている場合、分岐先への参照をサンプリングすることが可能な必要があります。他の命令への参照もサンプリングが可能かどうかは実装定義です。任意の命令への参照をサンプリング可能にすることをお勧めします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If an instruction writes to the CONTEXTIDR, it is UNPREDICTABLE whether the DBGCIDSR is set to the original or new value of CONTEXTIDR when a read of the DBGPCSR samples a subsequent instruction that occurs before the earliest of:

- the execution of an ISB instruction or an ISB operation
- the taking of an exception
- the execution of an exception return instruction.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

条件付き分岐命令で、条件コードチェックが失敗したときの分岐先は、条件付き分岐命令の次にある命令です。例外の分岐先は、例外ベクタアドレスです。

命令が CONTEXTIDR に書き込みを行ってから、DBGPCSR の読み出しによって、次のうちいずれかが行われる前に発生した、以後の命令がサンプリングされた場合、DBGCIDSR が CONTEXIDR の元の値に設定されるか、新しい値に設定されるかは予測不能です。

- ISB 命令の実行または ISB 操作
- 例外の取得
- 例外からの復帰命令の実行

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter C9
Performance Monitors

This chapter describes the performance monitors, that are a non-invasive debug component. It contains the following sections:

- *About the performance monitors* on page C9-2
- *Status in the ARM architecture* on page C9-4
- *Accuracy of the performance monitors* on page C9-5
- *Behavior on overflow* on page C9-6
- *Interaction with Security Extensions* on page C9-7
- *Interaction with trace* on page C9-8
- *Interaction with power saving operations* on page C9-9
- *CP15 c9 register map* on page C9-10
- *Access permissions* on page C9-12
- *Event numbers* on page C9-13.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 C9 章
パフォーマンスモニタ

本章では、パフォーマンスモニタについて説明します。これは非侵襲性デバッグコンポーネントです。本章は次のセクションから構成されています。

- *パフォーマンスモニタについて* P. C9-2
- *ARM アーキテクチャでのステータス* P. C9-4
- *パフォーマンスモニタの正確性* P. C9-5
- *オーバフロー時の動作* P. C9-6
- *セキュリティ拡張機能との相互動作* P. C9-7
- *トレースとの相互動作* P. C9-8
- *電力削減動作との相互動作* P. C9-9
- *CP15 c9 レジスタマップ* P. C9-10
- *アクセス許可* P. C9-12
- *イベント番号* P. C9-13

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.1 About the performance monitors

The basic organization of the performance monitors is:

- A cycle counter. This can be programmed to increment either on every cycle, or once every 64 cycles.
- A number of event counters. Each counter is configured to select the event that increments the counter. Space is provided in the architecture for up to 31 counters. The actual number of counters is IMPLEMENTATION DEFINED, and there is an identification mechanism for the counters.
- Controls for enabling the counters, resetting the counters, flagging overflows, and enabling interrupts on counter overflow.

  The cycle counter can be enabled independently of the event counters.

The counters are held in a set of registers that can be accessed in coprocessor space. This means the counters can be accessed from the operating system running on the processor, enabling a number of uses, including:

- dynamic compilation techniques
- energy management.

In addition, you can provide access to the counters from application code, if required. This enables applications to monitor their own performance with fine grain control without requiring operating system support. For example, an application might implement per-function performance monitoring.

There are many situations where performance monitoring features integrated into the processor are valuable for applications and for application development. When an operating system does not use the performance monitors itself, ARM recommends that it enables application code access to the performance monitors. However an implementation can choose not to implement any performance monitors.

To enable interaction with external monitoring, an implementation might consider additional enhancements, including:

- Providing a set of events, from which a selection can be exported onto a bus for use as external events. For very high frequency operation, this might introduce unacceptable timing requirements, but the bus could be interfaced to the trace macrocell or another closely coupled resource.
- Providing the ability to count external events. Here, again, there are clock frequency issues between the processor and the system. A suitable approach might be to edge-detect changes in the signals and to use those changes to increment a counter.

  This enhancement requires the processor to implement a set of external event input pins.
- Providing memory-mapped and external debug access to the performance monitor registers, to enable the counter resources to be used for system monitoring in systems where they are not used by the software running on the processor.

  Such access is not described in this manual. Contact ARM if you require more information about this option.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C9.1 パフォーマンスモニタについて

パフォーマンスモニタの基本構成は次のとおりです。

- サイクルカウンタ。すべてのサイクル、または 64 サイクルごとにインクリメントするようにプログラム可能です。
- 一連のイベントカウンタ。各カウンタは、そのカウンタをインクリメントするイベントを選択するように構成可能です。アーキテクチャには、最大 31 個のカウンタ用の領域が用意されています。カウンタの実際の数は実装定義で、識別機構が用意されています。
- カウンタの稼動、カウンタのリセット、およびオーバフローのフラグ付けを行い、カウンタのオーバフロー時の割り込みを可能にするコントローラ。

  サイクルカウンタは、イベントカウンタとは独立して稼動できます。

カウンタは一連のレジスタに保持され、コプロセッサ空間からアクセス可能です。つまり、プロセッサで実行されているオペレーティングシステムからカウンタにアクセスし、次のような各種の目的に使用できます。

- 動的なコンパイル技法
- 電力管理

さらに、必要であればアプリケーションコードからカウンタへのアクセスを提供することもできます。これによって、アプリケーションはオペレーティングシステムのサポートを必要とせずに、粒度の細かい制御を行って自分のパフォーマンスを監視できます。例えば、アプリケーションに機能ごとのパフォーマンス監視を実装することが可能です。

多くの状況で、プロセッサに組み込まれたパフォーマンス監視機能は、アプリケーション、およびアプリケーション開発にとって重要となります。オペレーティングシステムが自分でパフォーマンスモニタを使用しない場合でも、アプリケーションコードがパフォーマンスモニタへアクセスできるようにすることをお勧めします。ただし、実装にパフォーマンスモニタを一切実装しないことも選択できます。

外部の監視との相互動作を可能にするため、実装に次のような拡張機能を含めることもできます。

- 選択内容をバスにエクスポートし、外部イベントとして使用できるようにするための、一連のイベントを提供する。動作周波数が非常に高い場合、これによって許容できないタイミング要件が発生することがありますが、このバスはトレースマクロセルや他の密結合されたリソースに接続できます。
- 外部イベントのカウント機能を提供する。この場合も、プロセッサとシステムとの間でクロック周波数の問題が発生することがあります。適切な方法としては、信号の変化エッジを検出し、その変化を使用してカウンタをインクリメントします。

  この拡張機能には、プロセッサに一連の外部イベント入力ピンを実装する必要があります。
- プロセッサで実行されているソフトウェアによってカウンタが使用されないシステムにおいて、メモリマップおよび外部デバッグによるパフォーマンスモニタのレジスタへのアクセスを提供し、システムの監視にカウンタのリソースを使用できるようにする。

  このようなアクセスについては、本書では触れません。この手法の詳細については、ARM にお問い合わせ下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The set of events that might be monitored splits into:

- events that are likely to be consistent across many microarchitectures
- implementation specific events.

Therefore, this architecture defines a common set of events to be used across many microarchitectures, and a large space reserved for IMPLEMENTATION DEFINED events.

The full set of events for any given implementation is IMPLEMENTATION DEFINED, and there is no requirement to implement any of the common set of events. ARM recommends that ARMv7 processors implement as many of the events as are feasible given the architecture profile and microarchitecture of the implementation.

The event numbers of the common set of events are reserved for the specified events. In this set, a particular event number must either:

- be used for its assigned event
- not be used.

When an ARMv7 processor supports monitoring of an event that is assigned a number in the range allocated to the common set of events range, if possible it must use that number for the event. However, ARM might introduce additional event definitions in this range in future editions of this manual. Therefore software might encounter implementations where an event assigned a number in this range is monitored using an event number from the IMPLEMENTATION DEFINED range.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

監視対象とできるイベントの組は、次のように分割できます。

- 多くのマイクロアーキテクチャについて一貫している可能性が高いイベント
- 実装固有のイベント

したがって、このアーキテクチャでは、多くのマイクロアーキテクチャにわたって使用される共通のイベントの組と、実装定義イベント用に予約される大きな領域が定義されます。

特定の実装におけるイベントの完全な組は実装定義で、共通のイベントの組についても、いずれかを実装する必要条件はありません。ARMv7 プロセッサには、アーキテクチャのプロファイルと実装のマイクロアーキテクチャを考慮して、実用的な限り多くのイベントを実装することをお勧めします。

共通のイベントの組に対応するイベント番号は、指定のイベント用に予約されています。この組に含まれるイベント番号については、次のいずれかを行う必要があります。

- 割り当てられたイベント用に使用する。
- 使用しない。

ARMv7 プロセッサが、共通イベントの組に割り当てられている範囲内の番号に対応するイベントの監視をサポートする場合、可能であればそのイベントについてその番号を使用する必要があります。ただし、本書の将来のエディションで、この範囲に新たなイベントの定義が追加される可能性があります。このため、ソフトウェアでは、この範囲内の番号を割り当てられたイベントが、実装定義範囲内のイベント番号を使用して監視される実装が存在する可能性があることを考慮する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.2 Status in the ARM architecture

The status of the architecturally-defined performance monitors block is that it is an IMPLEMENTATION DEFINED space for ARMv7, but ARM recommends implementers to use the approach described here to implement the performance monitors.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C9.2 ARM アーキテクチャでのステータス

アーキテクチャで定義されたパフォーマンスモニタ ブロックのステータスは、ARMv7 では実装定義空間です。ただし、実装者はここで説明されている手法を使用してパフォーマンスモニタを実装することが推奨されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.3 Accuracy of the performance monitors

The performance monitors provide approximately accurate performance count information. To keep the implementation and validation cost low, a reasonable degree of inaccuracy in the counts is acceptable. There is no exact definition of *reasonable degree of inaccuracy*, but ARM recommends the following guidelines:

- Under normal operating conditions, the counters must present an accurate value of the count.
- In exceptional circumstances, such as changes in security state or other boundary conditions, it is acceptable for the count to be inaccurate.
- Under very unusual non-repeating pathological cases counts can be inaccurate. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in the count is vanishingly unlikely.

Note

An implementation must not introduce inaccuracies that can be triggered systematically by normal pieces of code that are running. For example, dropping a branch count in a loop due to the structure of the loop gives a systematic error that makes the count of branch behavior very inaccurate, and this is not reasonable. However, the dropping of a single branch count as the result of a rare interaction with an interrupt is acceptable.

The permitted inaccuracy limits the possible uses of the performance monitors. In particular, the point in a pipeline where the event counter is incremented is not defined relative to the point where a read of the event counters is made. This means that pipelining effects can cause some imprecision. An implementation must document any particular scenarios where significant inaccuracies are expected.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C9.3 パフォーマンスモニタの正確性

パフォーマンスモニタは、大まかに正確なパフォーマンスカウント情報を提供します。実装と検証のコストを削減するため、妥当なレベルでカウントが不正確になることは許容範囲です。*妥当なレベルの不正確性*の正確な定義はありませんが、次のガイドラインを推奨します。

- 通常の動作状況で、カウンタは正確なカウント値を提供する必要があります。
- 例外的な状況、例えばセキュリティ状態の変化や他の境界条件では、カウントが不正確になることは許容されます。
- 非常に例外的な、繰り返されない異常な条件では、カウントが不正確でも問題ありません。このような条件は、割り込みなどの非同期例外の結果として発生することが考えられ、カウントに体系的なエラーが発生することはほぼ考えられません。

注

実装で、コードの通常部分の実行により体系的にトリガされるような不正確性を導入することは許されません。例えば、ループ内の分岐カウントをドロップすると、ループの構造の関係で、分岐の動作のカウントを非常に不正確にする体系的なエラーが引き起こされることになり、これは妥当ではありません。ただし、割り込みとの稀な相互動作の結果として、単一の分岐カウントがドロップされることは許容されます。

許容される不正確性により、パフォーマンスモニタがどのような目的に使用可能であるかが制限されます。特に、パイプラインの中でイベントカウンタがインクリメントされる場所と、イベントカウンタの読み出しが行われる場所との相対関係は定義されていません。つまり、パイプラインの影響により多少の不正確性が発生する可能性があります。実装で、大きな不正確性が予測される特定のシナリオが存在する場合、そのことをドキュメントに記載する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.4 Behavior on overflow

On counter overflow:

- An overflow status flag is set to 1. See *c9, Overflow Flag Status Register (PMOVSR)* on page C10-110.
- An interrupt request is generated if the processor is configured to generate counter overflow interrupts. For details see *c9, Interrupt Enable Set Register (PMINTENSET)* on page C10-118 and *c9, Interrupt Enable Clear Register (PMINTENCLR)* on page C10-119.
- The counter wraps to zero and continues counting events. Counting continues as long as the counters are enabled, regardless of any overflows.

The counter always resets to zero and overflows after 32 bits of increment. To enable a more frequent generation of interrupt requests, the counters can be written to. For example, an interrupt handler might reset the overflowed counter to `0xFFFF0000` to generate another overflow interrupt after 16 bits of increment.

Note

The mechanism by which an interrupt request from the performance monitors generates an FIQ or IRQ exception is IMPLEMENTATION DEFINED.

The interrupt handler for the counter interrupt must cancel the interrupt by clearing the overflow flag.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C9.4 オーバフロー時の動作

カウンタのオーバフロー時には、次の動作が行われます。

- オーバフローステータス フラグが 1 にセットされます。*c9、オーバフローフラグステータス レジスタ (PMOVSR)* P. C10-110 を参照して下さい。
- カウンタオーバフロー割り込みを生成するようにプロセッサが構成されている場合、割り込み要求が生成されます。詳細については、*c9、割り込みイネーブルセット レジスタ (PMINTENSET)* P. C10-118 および *c9、割り込みイネーブルクリア レジスタ (PMINTENCLR)* P. C10-119 を参照して下さい。
- カウンタが 0 にラップアラウンドし、イベントのカウントが続行されます。カウントは、オーバフローの発生にかかわらず、カウンタが稼動している限り続行されます。

カウンタは常に 0 にリセットされ、32 ビットのインクリメントの後でオーバフローします。割り込み要求をより頻繁に生成できるようにするため、カウンタに書き込みを行うことができます。例えば、割り込みハンドラでオーバフローしたカウンタを `0xFFFF0000` にリセットすると、16 ビットのインクリメント後に新たなオーバフロー割り込みが生成されます。

注

パフォーマンスモニタからの割り込み要求により FIQ または IRQ 例外を生成する機構は実装定義です。

カウンタ割り込み用の割り込みハンドラは、オーバフローフラグをクリアして割り込みをキャンセルする必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.5 Interaction with Security Extensions

The performance monitors provide a non-invasive debug feature, and therefore are controlled by the non-invasive debug authentication signals. *About non-invasive debug authentication* on page C7-2 describes how non-invasive debug interacts with Security Extensions.

Performance monitors on page C7-6 describes the behavior of the performance monitors when:

- non-invasive debug is disabled
- the processor is in a mode or state where non-invasive debug is not permitted
- the processor is in Debug state.

Note

Additional controls in the PMCR can also disable the event counters and the PMCCNTR. Disabling the event counters and the PMCCNTR in the PMCR takes precedence over the authentication controls.

The performance monitor registers are Common registers, see *Common CP15 registers* on page B3-74. They are always accessible regardless of the values of the authentication signals and SUNIDEN. Authentication controls whether the counters count events, not to control access to the performance monitor registers.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.5 セキュリティ拡張機能との相互動作

パフォーマンスモニタは非侵襲性デバッグ機能を提供するため、非侵襲性デバッグ認証信号によって制御されます。非侵襲性デバッグとセキュリティ拡張機能との相互動作については、*非侵襲性デバッグの認証について*P. C7-2 を参照して下さい。

次の状況でのパフォーマンスモニタの動作については、*パフォーマンスモニタ*P. C7-6 を参照して下さい。

- 非侵襲性デバッグが非稼動である。
- プロセッサが、非侵襲性デバッグが許可されないモードまたは状態である。
- プロセッサがデバッグ状態である。

注

PMCR の追加コントロールも、イベントカウンタと PMCCNTR を非稼動にすることがあります。PMCR によるイベントカウンタと PMCCNTR の稼動停止は、認証コントロールより優先されます。

パフォーマンスモニタのレジスタは共通レジスタです。*共通の CP15 レジスタ*P. B3-74 を参照して下さい。これらのレジスタには、認証信号と SUNIDEN の値にかかわらず、常にアクセス可能です。認証により制御されるのは、カウンタによってイベントがカウントされるかどうかで、パフォーマンスモニタのレジスタへのアクセスは制御されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.6 Interaction with trace

It is IMPLEMENTATION DEFINED whether counter events are exported to a trace macrocell or other external monitoring agents to provide triggering information. The form of the exporting is also IMPLEMENTATION DEFINED. If implemented, this exporting might be enabled as part of the performance monitoring control functionality.

Similarly, ARM recommends system designers to include a mechanism for importing a set of external events to be counted, but such a feature is IMPLEMENTATION DEFINED. When implemented, this feature enables the trace module to pass in events to be counted.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C9.6 トレースとの相互動作

トリガに関する情報を提供するため、カウンタイベントがトレースマクロセルや他の外部監視エージェントにエクスポートされるかどうかは実装定義です。エクスポートの形式も実装定義です。このエクスポートが実装されている場合、パフォーマンスモニタの制御機能の一部として稼動状態になる可能性があります。

同様に、システム設計者はカウントされる外部イベントの組をインポートする機構を含めることが推奨されます。ただし、このような機能は実装定義です。この機能が実装されている場合、トレースモジュールがカウントされるイベントを渡すことが可能になります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.7 Interaction with power saving operations

All counters are subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C9.7 電力削減動作との相互動作

すべてのカウンタは、クロック周波数の変化の影響を受けます。これには、WFI および WFE 命令によるクロック停止も含まれます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.8 CP15 c9 register map

The performance monitor registers are mapped into part of the CP15 register map. The registers are described in *Performance monitor registers* on page C10-105.

Figure C9-1 shows the CP15 c9 encodings for the recommended performance monitor registers, and the reserved encodings for IMPLEMENTATION DEFINED performance monitors:

Figure C9-1 Recommended CP15 performance monitor registers

Table C9-1 lists the instructions used to access the recommended performance monitor registers.

Table C9-1 Recommended performance monitor registers

Instruction[a]	Description or notes
`MRC p15,0,<Rt>,c9,c12,0` `MCR p15,0,<Rt>,c9,c12,0`	*c9, Performance Monitor Control Register (PMCR)* on page C10-105.
`MRC p15,0,<Rt>,c9,c12,1` `MCR p15,0,<Rt>,c9,c12,1`	*c9, Count Enable Set Register (PMCNTENSET)* on page C10-108.
`MRC p15,0,<Rt>,c9,c12,2` `MCR p15,0,<Rt>,c9,c12,2`	*c9, Count Enable Clear Register (PMCNTENCLR)* on page C10-109.
`MRC p15,0,<Rt>,c9,c12,3` `MCR p15,0,<Rt>,c9,c12,3`	*c9, Overflow Flag Status Register (PMOVSR)* on page C10-110.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.8 CP15 c9 レジスタマップ

パフォーマンスモニタのレジスタは、CP15 レジスタマップの一部としてマッピングされます。これらのレジスタについては、*パフォーマンスモニタ レジスタ* P. C10-105 を参照して下さい。

推奨されるパフォーマンスモニタ レジスタの CP15 c9 エンコードと、実装定義パフォーマンスモニタ用に予約されているエンコードを、図 C9-1 に示します。

図 C9-1 推奨される CP15 パフォーマンスモニタ レジスタ

推奨されるパフォーマンスモニタ レジスタにアクセスするために使用される命令を、表 C9-1 に示します。

表 C9-1 推奨されるパフォーマンスモニタ レジスタ

命令[a]	説明または注
`MRC p15,0,<Rt>,c9,c12,0` `MCR p15,0,<Rt>,c9,c12,0`	*c9、パフォーマンスモニタ制御レジスタ (PMCR)* P. C10-105
`MRC p15,0,<Rt>,c9,c12,1` `MCR p15,0,<Rt>,c9,c12,1`	*c9、カウントイネーブルセット レジスタ (PMCNTENSET)* P. C10-108
`MRC p15,0,<Rt>,c9,c12,2` `MCR p15,0,<Rt>,c9,c12,2`	*c9、カウントイネーブルクリア レジスタ (PMCNTENCLR)* P. C10-109
`MRC p15,0,<Rt>,c9,c12,3` `MCR p15,0,<Rt>,c9,c12,3`	*c9、オーバフローフラグステータス レジスタ (PMOVSR)* P. C10-110

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C9-1 Recommended performance monitor registers (continued)

Instruction[a]	Description or notes
`MRC p15,0,<Rt>,c9,c12,4`	UNPREDICTABLE. PMSWINC is a write-only register.
`MCR p15,0,<Rt>,c9,c12,4`	*c9, Software Increment Register (PMSWINC)* on page C10-112.
`MRC p15,0,<Rt>,c9,c12,5` `MCR p15,0,<Rt>,c9,c12,5`	*c9, Event Counter Selection Register (PMSELR)* on page C10-113.
`MRC p15,0,<Rt>,c9,c13,0` `MCR p15,0,<Rt>,c9,c13,0`	*c9, Cycle Count Register (PMCCNTR)* on page C10-114.
`MRC p15,0,<Rt>,c9,c13,1` `MCR p15,0,<Rt>,c9,c13,1`	*c9, Event Type Select Register (PMXEVTYPER)* on page C10-115.
`MRC p15,0,<Rt>,c9,c13,2` `MCR p15,0,<Rt>,c9,c13,2`	*c9, Event Count Register (PMXEVCNTR)* on page C10-116.
`MRC p15,0,<Rt>,c9,c14,0` `MCR p15,0,<Rt>,c9,c14,0`	*c9, User Enable Register (PMUSERENR)* on page C10-117.
`MRC p15,0,<Rt>,c9,c14,1` `MCR p15,0,<Rt>,c9,c14,1`	*c9, Interrupt Enable Set Register (PMINTENSET)* on page C10-118.
`MRC p15,0,<Rt>,c9,c14,2` `MCR p15,0,<Rt>,c9,c14,2`	*c9, Interrupt Enable Clear Register (PMINTENCLR)* on page C10-119.

a. CP15 c9 encodings with CRm == {c12-c14} not listed in the table are reserved. For details of the behavior of accesses to these encodings see *Unallocated CP15 encodings* on page B3-69.

C9.8.1 Power domains and performance monitor registers reset

For ARMv7 implementations, ARM recommends that performance monitors are implemented as part of the core power domain, not as part of a separate debug power domain. There is no interface to access the performance monitor registers when the core power domain is powered down.

The performance monitor registers must be set to their reset values on a processor reset by **nSYSPORESET**, **nCOREPORESET** or **nRESET**. Performance monitor registers are not changed by a debug logic reset by **PRESETDBGn**.

For more information about the reset scheme recommended for a v7 Debug implementation see *Recommended reset scheme for v7 Debug* on page C6-16.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C9-1 推奨されるパフォーマンスモニタ レジスタ（続き）

命令[a]	説明または注
`MRC p15,0,<Rt>,c9,c12,4`	予測不能。PMSWINC は書き込み専用レジスタです。
`MCR p15,0,<Rt>,c9,c12,4`	*c9、ソフトウェアインクリメント レジスタ (PMSWINC)* P. C10-112
`MRC p15,0,<Rt>,c9,c12,5` `MCR p15,0,<Rt>,c9,c12,5`	*c9、イベントカウンタ選択レジスタ (PMSELR)* P. C10-113
`MRC p15,0,<Rt>,c9,c13,0` `MCR p15,0,<Rt>,c9,c13,0`	*c9、サイクルカウント レジスタ (PMCCNTR)* P. C10-114
`MRC p15,0,<Rt>,c9,c13,1` `MCR p15,0,<Rt>,c9,c13,1`	*c9、イベントタイプ選択レジスタ (PMXEVTYPER)* P. C10-115
`MRC p15,0,<Rt>,c9,c13,2` `MCR p15,0,<Rt>,c9,c13,2`	*c9、イベントカウント レジスタ (PMXEVCNTR)* P. C10-116
`MRC p15,0,<Rt>,c9,c14,0` `MCR p15,0,<Rt>,c9,c14,0`	*c9、ユーザイネーブル レジスタ (PMUSERENR)* P. C10-117
`MRC p15,0,<Rt>,c9,c14,1` `MCR p15,0,<Rt>,c9,c14,1`	*c9、割り込みイネーブルセット レジスタ (PMINTENSET)* P. C10-118
`MRC p15,0,<Rt>,c9,c14,2` `MCR p15,0,<Rt>,c9,c14,2`	*c9、割り込みイネーブルクリア レジスタ (PMINTENCLR)* P. C10-119

a. CP15 c9 エンコードで CRm == {c12 ～ c14} のもののうち、この表に記載されていないものは予約されています。これらのエンコードに対するアクセスの動作の詳細については、*未割り当ての CP15 エンコード* P. B3-69 を参照して下さい。

C9.8.1 電力ドメインとパフォーマンスモニタ レジスタのリセット

ARMv7 実装では、パフォーマンスモニタを独立したデバッグ電力ドメインの一部ではなく、コア電力ドメインの一部として実装することをお勧めします。コア電力ドメインの電力がオフになっているとき、パフォーマンスモニタ レジスタにアクセスするためのインタフェースは存在しません。

パフォーマンスモニタ レジスタは、nSYSPORESET、nCOREPORESET、または nRESET によるプロセッサのリセット時に、リセット時の値に設定される必要があります。パフォーマンスモニタ レジスタは、PRESETDBGn によるデバッグロジック リセットでは変更されません。

v7 デバッグ実装で推奨されるリセット方式の詳細については、*v7 デバッグで推奨されるリセット方式* P. C6-16 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.9 Access permissions

Normally the performance monitor registers are accessible from privileged modes only. Setting the PMUSERENR.EN flag to 1 permits access from User mode code, for example for instrumentation and profiling purposes, see *c9, User Enable Register (PMUSERENR)* on page C10-117. However, the PMUSERENR does not provide access to the registers that control interrupt generation.

Table C9-2 Performance monitor access permissions

Register	Operation	Access from a privileged mode	Access from User mode[a]	
			PMUSERENR.EN == 0	PMUSERENR.EN == 1
PMCR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMCNTENSET	MRC or MCR	Proceed	UNDEFINED	Proceed
PMCNTENCLR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMOVSR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMSWINC	MRC	UNPREDICTABLE	UNDEFINED	UNPREDICTABLE
	MCR	Proceed	UNDEFINED	Proceed
PMSELR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMCCNTR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMXEVTYPER	MRC or MCR	Proceed	UNDEFINED	Proceed
PMXEVCNTR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMUSERENR[a]	MRC	Proceed	Proceed	Proceed
	MCR	Proceed	UNDEFINED	UNDEFINED
PMINTENSET	MRC or MCR	Proceed	UNDEFINED	UNDEFINED
PMINTENCLR	MRC or MCR	Proceed	UNDEFINED	UNDEFINED
Reserved[b]	MRC or MCR	UNPREDICTABLE	UNDEFINED	UNDEFINED

a. For details of the EN flag see *c9, User Enable Register (PMUSERENR)* on page C10-117.
b. All the registers marked as reserved in Table C9-1 on page C9-10.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C9.9 アクセス許可

通常は、パフォーマンスモニタ レジスタは特権モードからのみアクセス可能です。PMUSERENR.EN フラグを 1 にセットすると、ユーザモードのコードからのアクセスが許可され、計装やプロファイリングなどの目的に使用できます。*c9、ユーザイネーブル レジスタ (PMUSERENR)* P. C10-117 を参照して下さい。ただし、PMUSERENR では、割り込み生成を制御するレジスタへのアクセスは提供されません。

表 C9-2 パフォーマンスモニタのアクセス許可

レジスタ	動作	特権モードからのアクセス	ユーザモードからのアクセス[a]	
			PMUSERENR.EN == 0	PMUSERENR.EN == 1
PMCR	MRC または MCR	実行	未定義	実行
PMCNTENSET	MRC または MCR	実行	未定義	実行
PMCNTENCLR	MRC または MCR	実行	未定義	実行
PMOVSR	MRC または MCR	実行	未定義	実行
PMSWINC	MRC	予測不能	未定義	予測不能
	MCR	実行	未定義	実行
PMSELR	MRC または MCR	実行	未定義	実行
PMCCNTR	MRC または MCR	実行	未定義	実行
PMXEVTYPER	MRC または MCR	実行	未定義	実行
PMXEVCNTR	MRC または MCR	実行	未定義	実行
PMUSERENR[a]	MRC	実行	実行	実行
	MCR	実行	未定義	未定義
PMINTENSET	MRC または MCR	実行	未定義	未定義
PMINTENCLR	MRC または MCR	実行	未定義	未定義
予約[b]	MRC または MCR	予測不能	未定義	未定義

a. EN フラグの詳細については、*c9、ユーザイネーブル レジスタ (PMUSERENR)* P. C10-117 を参照して下さい。
b. P. C9-10 の表 C9-1 で予約としてマークされているすべてのレジスタ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.10 Event numbers

The event numbers are described in the following subsections:

- *Common feature event numbers*
- *Implementation defined feature event numbers* on page C9-16.

C9.10.1 Common feature event numbers

For the common features, normally the counters must increment only once for each event. Exceptions to this rule are stated in the individual definitions.

In these definitions, the term *architecturally executed* means that the instruction flow is such that the counted instruction would have been executed in a simple sequential execution model.

Note

An instruction is architecturally executed if the behavior of the program on the processor is consistent with the instruction having been executed on a simple execution model of the architecture. Therefore an instruction that has been executed and retired is defined to be *architecturally executed*. In processors that perform speculative execution, an instruction is not architecturally executed if the results of the speculative execution are discarded. Where an instruction has no visible effect, for example, a `NOP`, the point where the instruction is retired is IMPLEMENTATION DEFINED.

The common feature event number assignments are:

`0x00` Software increment. The register is incremented only on writes to the Software Increment Register. For details see *c9, Software Increment Register (PMSWINC)* on page C10-112.

`0x01` Instruction fetch that causes a refill of at least the level of instruction or unified cache closest to the processor. Each instruction fetch that causes a refill from outside the cache is counted. Accesses that do not cause a new cache refill, but are satisfied from refilling data of a previous miss, are not counted. Where an instruction fetch fetches multiple instructions, the fetch counts a single event.

CP15 cache maintenance operations do not count as events.

This counter increments on speculative instruction fetches as well as on fetches of instructions that reach execution.

`0x02` Instruction fetch that causes a TLB refill of at least the level of TLB closest to the processor. Each instruction fetch that causes an access to a level of memory system due to a translation table walk or an access to another level of TLB caching is counted.

CP15 TLB maintenance operations do not count as events.

This counter increments on speculative instruction fetches as well as on fetches of instructions that reach execution.

`0x03` Memory Read or Write operation that causes a refill of at least the level of data or unified cache closest to the processor. Each memory read from or write to that causes a refill from outside the cache is counted. Accesses that do not cause a new cache refill, but are satisfied

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C9.10 イベント番号

イベント番号については、次のサブセクションを参照して下さい。

- *共通機能のイベント番号*
- *実装定義機能のイベント番号* P. C9-16

C9.10.1 共通機能のイベント番号

共通機能については、カウンタは通常、各イベントごとに 1 回だけインクリメントされる必要があります。この規則の例外は、それぞれの定義に記載されています。

これらの定義で、*アーキテクチャ的に実行される*という用語は、命令フローが、カウント対象の命令が単純なシーケンシャル実行モデルで実行されたと仮定した場合と同様であることを意味します。

注

プロセッサ上のプログラムの動作が、その命令がアーキテクチャの単純な実行モデルで実行されたと仮定した場合と整合していれば、命令はアーキテクチャ的に実行されます。このため、実行されて終了した命令は、*アーキテクチャ的に実行された*ものと定義されます。投機実行を行うプロセッサで、投機実行の結果が破棄された場合、命令はアーキテクチャ的に実行されていません。命令に可視の影響がない、例えば `NOP` の場合、どの時点で命令が終了するのかは実装定義です。

共通機能のイベント番号割り当ては次のとおりです。

`0x00` ソフトウェアインクリメント。レジスタは、ソフトウェアインクリメント レジスタへの書き込みによってのみインクリメントされます。詳細については、*c9、ソフトウェアインクリメント レジスタ (PMSWINC)* P. C10-112 を参照して下さい。

`0x01` 命令フェッチで、最低でもプロセッサに最も近いレベルの命令または統一キャッシュのリフィルを引き起こすもの。命令フェッチのうち、キャッシュの外側からのリフィルを引き起こすものはすべてカウントされます。新しいキャッシュリフィルを引き起こさず、以前のミスによるデータのリフィルによって満たされるものはカウントされません。命令フェッチによって複数の命令がフェッチされる場合、そのフェッチは単一のイベントとしてカウントされます。

CP15 キャッシュ保守操作は、イベントとしてカウントされません。

このカウンタは、投機的な命令フェッチでも、実行段階に到達する命令のフェッチと同様にインクリメントされます。

`0x02` 命令フェッチで、最低でもプロセッサに最も近いレベルの TLB のリフィルを引き起こすもの。命令フェッチのうち、変換テーブルウォークや別レベルの TLB キャッシュへのアクセスにより、あるレベルのメモリシステムへのアクセスを引き起こすものは、すべてカウントされます。

CP15 TLB 保守操作は、イベントとしてカウントされません。

このカウンタは、投機的な命令フェッチでも、実行段階に到達する命令のフェッチと同様にインクリメントされます。

`0x03` メモリ読み出しまたは書き込み操作で、最低でもプロセッサに最も近いレベルのデータまたは統一キャッシュのリフィルを引き起こすもの。メモリ読み出しまたは書き込みで、キャッシュの外側からのリフィルを引き起こすものはすべてカウ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

from refilling data of a previous miss are not counted. Each access to a cache line that causes a new linefill is counted, including the multiple accesses of load or store multiples, including PUSH and POP. Write-Through writes that hit in the cache do not cause a linefill and so are not counted.

CP15 cache maintenance operations do not count as events.

This counter increments on speculative memory accesses as well as for memory accesses that are explicitly made by instructions.

0x04 Memory Read or Write operation that causes a cache access to at least the level of data or unified cache closest to the processor. Each access to a cache line is counted including the multiple accesses of instructions such as LDM or STM.

CP15 cache maintenance operations do not count as events.

This counter increments on speculative memory accesses as well as for memory accesses that are explicitly made by instructions.

0x05 Memory Read or Write operation that causes a TLB refill of at least the level of TLB closest to the processor. Each memory read or write operation that causes a translation table walk or an access to another level of TLB caching is counted.

CP15 TLB maintenance operations do not count as events.

This counter increments on speculative memory accesses as well as for memory accesses that are explicitly made by instructions.

0x06 Memory-reading instruction architecturally executed. This counter increments for every instruction that explicitly read data, including SWP.

This counter does not increment for a conditional instruction that fails its condition code check.

0x07 Memory-writing instruction architecturally executed. The counter increments for every instruction that explicitly wrote data, including SWP.

This counter does not increment for a Store-Exclusive instruction that fails, or for a conditional instruction that fails its condition code check.

0x08 Instruction architecturally executed. This counter counts for all instructions, including conditional instructions that fail their condition code check.

0x09 Exception taken. This counts for each exception taken.

Note

This event number counts the processor exceptions described in *Exceptions* on page B1-30. It does not count floating-point exceptions or ThumbEE null and index checks.

0x0A Exception return architecturally executed. This counts the exception return instructions described in *Exception return* on page B1-38.

This counter does not increment for a conditional instruction that fails its condition code check.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ントされます。新しいキャッシュリフィルを引き起こさず、以前のミスによるデータのリフィルによって満たされるものはカウントされません。キャッシュラインへのアクセスが新しいラインフィルを引き起こす場合、PUSH や POP など複数ロード / ストアによる複数回のアクセスも含めて、すべてカウントされます。ライトスルーの書き込みがキャッシュにヒットした場合、ラインフィルが引き起こされないため、カウントされません。

CP15 キャッシュ保守操作は、イベントとしてカウントされません。

このカウンタは、投機的なメモリアクセスでも、命令によって明示的に行われたメモリアクセスと同様にインクリメントされます。

0x04 メモリ読み出しまたは書き込み操作で、最低でもプロセッサに最も近いレベルのデータまたは統一キャッシュへのキャッシュアクセスを引き起こすもの。キャッシュラインへのアクセスは、LDM や STM などの命令による複数回のアクセスも含めて、すべてカウントされます。

CP15 キャッシュ保守操作は、イベントとしてカウントされません。

このカウンタは、投機的なメモリアクセスでも、命令によって明示的に行われたメモリアクセスと同様にインクリメントされます。

0x05 メモリ読み出しまたは書き込み操作で、最低でもプロセッサに最も近いレベルの TLB のリフィルを引き起こすもの。メモリ読み出しまたは書き込み操作で、変換テーブルウォークや別レベルの TLB キャッシュへのアクセスを引き起こすものは、すべてカウントされます。

CP15 TLB 保守操作は、イベントとしてカウントされません。

このカウンタは、投機的なメモリアクセスでも、命令によって明示的に行われたメモリアクセスと同様にインクリメントされます。

0x06 アーキテクチャ的に実行されたメモリ読み出し命令。このカウンタは、SWP も含めて、明示的にデータを読み出すすべての命令でインクリメントされます。

このカウンタは、条件付き命令が条件コードチェックに失敗した場合はインクリメントされません。

0x07 アーキテクチャ的に実行されたメモリ書き込み命令。このカウンタは、SWP も含めて、明示的にデータを書き込むすべての命令でインクリメントされます。

排他的ストア命令が失敗した場合や、条件付き命令が条件コードチェックに失敗した場合、このカウンタはインクリメントされません。

0x08 アーキテクチャ的に実行された命令。このカウンタは、条件付き命令が条件コードチェックに失敗した場合も含めて、すべての命令をカウントします。

0x09 取得された例外。取得された例外すべてがカウントされます。

注

このイベント番号は、*例外* P. B1-30 に説明されているプロセッサ例外をカウントします。浮動小数点例外や、ThumbEE の NULL およびインデクスチェックはカウントされません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0x0B Instruction that writes to the CONTEXTIDR architecturally executed.

This counter does not increment for a conditional instruction that fails its condition code check.

0x0C Software change of PC, except by an exception, architecturally executed.

This counter does not increment for a conditional instruction that fails its condition code check.

0x0D Immediate branch architecturally executed:

- `B{L} <label>`
- `BLX <label>`
- `CB{N}Z <Rn>,<label>`
- `HB{L} #HandlerId` (ThumbEE state only)
- `HB{L}P #<imm>, #HandlerId` (ThumbEE state only).

This counter counts for all immediate branch instructions that are architecturally executed, including conditional instructions that fail their condition code check.

0x0E Procedure return, other than exception return, architecturally executed:

- `BX R14`
- `MOV PC,LR`
- `POP {…,PC}`
- `LDR PC,[SP],#offset`
- `LDMIA R9!,{…,PC}` (ThumbEE state only)
- `LDR PC,[R9],#offset` (ThumbEE state only).

This counter does not increment for a conditional instruction that fails its condition code check.

Note

Only these instructions are counted as procedure returns. For example, the following are not counted as procedure return instructions:

- `BX R0` (Rm != R14)
- `MOV PC,R0` (Rm != R14)
- `LDM SP,{…,PC}` (writeback not specified)
- `LDR PC,[SP,#offset]` (wrong addressing mode).

0x0F Unaligned access architecturally executed. This counts each instruction that is an access to an unaligned address. That is, the instruction either triggered an unaligned fault, or would have done so if the CPSR.A bit had been 1.

This counter does not increment for a conditional instruction that fails its condition code check.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0x0A アーキテクチャ的に実行された、例外からの復帰。*例外からの復帰* P. B1-38 に説明されている例外からの復帰命令がカウントされます。

このカウンタは、条件付き命令が条件コードチェックに失敗した場合はインクリメントされません。

0x0B CONTEXTIDR に書き込みを行う、アーキテクチャ的に実行された命令。

このカウンタは、条件付き命令が条件コードチェックに失敗した場合はインクリメントされません。

0x0C アーキテクチャ的に実行された、ソフトウェアによる PC の変更。ただし、例外によるものは除外されます。

このカウンタは、条件付き命令が条件コードチェックに失敗した場合はインクリメントされません。

0x0D アーキテクチャ的に実行された、イミディエート値による分岐。

- `B{L} <label>`
- `BLX <label>`
- `CB{N}Z <Rn>,<label>`
- `HB{L} #HandlerId`（ThumbEE 状態のみ）
- `HB{L}P #<imm>, #HandlerId`（ThumbEE 状態のみ）

このカウンタは、条件付き命令が条件コードチェックに失敗した場合も含めて、アーキテクチャ的に実行された、イミディエート値による分岐命令をすべてカウントします。

0x0E 例外からの復帰を除く、アーキテクチャ的に実行されたプロシージャからの復帰。

- `BX R14`
- `MOV PC,LR`
- `POP {…,PC}`
- `LDR PC,[SP],#offset`
- `LDMIA R9!,{...,PC}`（ThumbEE 状態のみ）
- `LDR PC,[R9],#offset`（ThumbEE 状態のみ）

このカウンタは、条件付き命令が条件コードチェックに失敗した場合はインクリメントされません。

注

プロシージャからの復帰としてカウントされるのは、これらの命令のみです。例えば、次の命令はプロシージャからの復帰命令としてカウントされません。

- `BX R0` (Rm != R14)
- `MOV PC,R0` (Rm != R14)
- `LDM SP,{...,PC}`（ライトバックが指定されていない）
- `LDR PC,[SP,#offset]`（アドレシングモードが間違っている）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

`0x10` Branch mispredicted or not predicted. This counts for each correction to the predicted program flow that occurs because of a misprediction from, or no prediction from, the program flow prediction resources and that relates to instructions that the program flow prediction resources are capable of predicting.

`0x11` Cycle count. The register is incremented on every cycle.

Note

Unlike PMCCNTR, this count is not affected by PMCR.DP, PMCR.D or PMCR.C:

- The counter is not incremented in prohibited regions, so is not affected by PMCR.DP.
- The counter increments on every cycle, regardless of the setting of PMCR.D.
- The counter is reset when event counters are reset by PMCR.P, never by PMCR.C.

`0x12` Branch or other change in program flow that could have been predicted by the branch prediction resources of the processor.

`0x13-0x3F` Reserved.

C9.10.2 IMPLEMENTATION DEFINED feature event numbers

For IMPLEMENTATION DEFINED feature numbers, the counters are defined to either:

- increment only once for each event
- count the duration for which an event occurs

This property is defined individually for each feature.

ARM recommends implementers to establish house styles for the IMPLEMENTATION DEFINED events, with common definitions, and common count numbers, applied to all the processors they implement. In general, the recommended approach is for standardization across implementations with common features. However, ARM recognizes that attempting to standardize the encoding of microarchitectural features across too wide a range of implementations is not productive.

ARM strongly recommends that at least the following classes of event are identified in the IMPLEMENTATION DEFINED events:

- Cumulative duration of stalls due to the holes in the instruction availability, separating out counts for key buffering points that might exist.
- Cumulative duration of stalls due to data dependent stalling, separating out counts for key dependency classes that might exist.
- Cumulative duration of stalls due to unavailability of execution resources (including write buffers, for example), separating out counts for key resources that might exist.
- Missed superscalar issue opportunities, if relevant, separating out counts for key classes of issue that might exist.
- Miss rates for different levels of caches and TLB.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

`0x0F` アーキテクチャ的に実行されたアンアラインドアクセス。アンアラインドアドレスにアクセスする命令がすべてカウントされます。つまり、アンアラインドフォールトをトリガしたか、CPSR.A ビットが 1 であればトリガする命令です。

このカウンタは、条件付き命令が条件コードチェックに失敗した場合はインクリメントされません。

`0x10` 予測に失敗したか、予測されなかった分岐。プログラムフローの予測リソースにより行われた予測が失敗した、または予測が行われなかったために、プログラムフロー予測リソースが予測可能である命令に関連している、予測されたプログラムフローが修正されるごとにカウントされます。

`0x11` サイクルカウント。このレジスタは、各サイクルごとにインクリメントされます。

注

PMCCNTR とは異なり、このカウントは PMCR.DP、PMCR.D、PMCR.C の影響を受けません。

- カウンタは、禁止領域ではインクリメントされないため、PMCR.DP の影響を受けません。
- カウンタは、PMCR.D の設定にかかわらず、すべてのサイクルでインクリメントされます。
- カウンタは、PMCR.P によりイベントカウンタがリセットされたときにリセットされ、PMCR.C によりリセットされることはありません。

`0x12` プロセッサの分岐予測リソースによって予測が可能であった、分岐または他のプログラムフロー変更。

`0x13` 〜 `0x3F` 予約

C9.10.2 実装定義機能のイベント番号

実装定義機能の番号については、カウンタは次のいずれかに定義されます。

- 各イベントごとに 1 回だけインクリメントされる
- イベントが発生していた期間をカウントする

このプロパティは、各機能について個別に定義されます。

実装定義イベントについては、実装者が自社の形式を確立し、実装するすべてのプロセッサに対して共通の定義と共通のカウント番号を適用することを推奨します。一般に、すべての実装にわたって共通の機能を標準化することが推奨される方法です。ただし、非常に広範な実装にわたってマイクロアーキテクチャ機能のエンコードの標準化を試みるのは生産的でないことも ARM は理解しています。

最低でも、次のクラスのイベントは実装定義イベントとして識別することを強く推奨します。

- 命令の可用性の穴によって生じたストールの累積時間。存在する可能性がある主要なバッファ点についてカウントを分離します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- Transaction counts on external buses.
- External events passed into the processor via an IMPLEMENTATION DEFINED mechanism. Typically this involves counting the number of cycles for which the signal is asserted using the duration count option.
- Cumulative duration for which the CPSR.I and CPSR.F interrupt mask bits are set to 1.
- Any other microarchitectural features that the implementer considers it valuable to count.

IMPLEMENTATION DEFINED feature numbers are `0x40` to `0xFF`.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- データ依存のストールによって生じたストールの累積時間。存在する可能性がある主要な依存性クラスについてカウントを分離します。
- 実行リソース（ライトバッファなどを含む）が利用できないために生じたストールの累積時間。存在する可能性がある主要なリソースについてカウントを分離します。
- スーパースカラ発行機会の失敗。該当する場合は、存在する可能性がある主要な発行のクラスについてカウントを分離します。
- 各レベルのキャッシュと TLB のミス率
- 外部バスのトランザクションカウント
- 実装定義の機構によりプロセッサに渡される外部イベント。一般に、このカウントには、期間カウントオプションを使用して、信号がアサートされるサイクル数をカウントする動作が関与します。
- CPSR.I および CPSR.F 割り込みマスクビットが 1 にセットされていた累積期間
- 実装者が、カウントすることが重要であると考える、他の任意のマイクロアーキテクチャ機能。

実装定義の機能の番号は `0x40` 〜 `0xFF` です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Chapter C10
Debug Registers Reference

This chapter gives a reference description of the debug registers. See *Debug register map* on page C6-18 for a list of all the debug registers.

This chapter contains the following sections:

- *Accessing the debug registers* on page C10-2
- *Debug identification registers* on page C10-3
- *Control and status registers* on page C10-10
- *Instruction and data transfer registers* on page C10-40
- *Software debug event registers* on page C10-48
- *OS Save and Restore registers, v7 Debug only* on page C10-75
- *Memory system control registers* on page C10-80
- *Management registers, ARMv7 only* on page C10-88
- *Performance monitor registers* on page C10-105.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 C10 章
デバッグレジスタのインタフェース

本章では、デバッグレジスタの参照用の説明を行います。デバッグレジスタの一覧については、*デバッグレジスタのマップ* P. C6-18 を参照して下さい。

本章は次のセクションから構成されています。

- *デバッグレジスタへのアクセス* P. C10-2
- *デバッグ識別レジスタ* P. C10-3
- *制御およびステータスレジスタ* P. C10-10
- *命令およびデータ転送レジスタ* P. C10-40
- *ソフトウェアデバッグイベント レジスタ* P. C10-48
- *OS 保存および復元レジスタ（v7 デバッグのみ）* P. C10-75
- *メモリシステム制御レジスタ* P. C10-80
- *管理レジスタ（ARMv7 のみ）* P. C10-88
- *パフォーマンスモニタ レジスタ* P. C10-105

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.1 Accessing the debug registers

In this chapter:

- The debug registers are numbered sequentially from 0 to 1023.
- The register offsets refer to the offsets in the v7 Debug memory-mapped or external debug interface. The locations of these registers in the ARMv6 external debug interface might differ.

There is a standard mapping from debug register number to coprocessor instructions in the Extended CP14 interface, see *Extended CP14 interface* on page C6-33. The register numbers and offsets for the DBGDSCR, DBGDTRRX, and DBGDTRTX Registers apply only to the external view of that register. For more information, see *Internal and external views of the DBGDSCR and the DCC registers* on page C6-21.

Note

- The recommended v7 Debug external debug interface is described in *ARM Debug Interface v5 Architecture Specification*.
- Contact ARM if you require details of the ARMv6 recommended external debug interface.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C10.1 デバッグレジスタへのアクセス

本章の記述には、次の前提が適用されます。

- デバッグレジスタには、順番に 0 ～ 1023 までの番号が付けられています。
- レジスタオフセットは、v7 デバッグのメモリマップされた、または外部デバッグインタフェースでのオフセットを指します。ARMv6 外部デバッグインタフェースでは、これらのレジスタの位置は異なる可能性があります。

デバッグレジスタ番号から、拡張 CP14 インタフェースのコプロセッサ命令への標準のマッピングについては、*拡張 CP14 インタフェース* P. C6- 33 を参照して下さい。DBGDSCR、DBGDTRRX、DBGDTRTX レジスタのレジスタ番号とオフセットは、これらのレジスタの外部ビューにのみ適用されます。詳細については、*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6- 21 を参照して下さい。

注

- 推奨される v7 デバッグの外部デバッグインタフェースについては、『*ARM デバッグインタフェース v5、アーキテクチャ仕様書*』を参照して下さい。
- ARMv6 での推奨される外部デバッグインタフェースの詳細については、ARM にお問い合わせ下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.2 Debug identification registers

This section contains the following subsections:

- *Debug ID Register (DBGDIDR)*
- *Debug Device ID Register (DBGDEVID)* on page C10-6
- *Debug ROM Address Register (DBGDRAR)* on page C10-7
- *Debug Self Address Offset Register (DBGDSAR)* on page C10-8.

C10.2.1 Debug ID Register (DBGDIDR)

The Debug ID Register, DBGDIDR, specifies:

- which version of the Debug architecture is implemented
- some features of the debug implementation.

The DBGDIDR is:

- debug register 0, at offset `0x000`
- a read-only register
- required on all versions of the Debug architecture from v6 Debug onwards
- when the Security Extensions are implemented, a Common register.

The format of the DBGDIDR is:

WRPs, bits [31:28]

The number of *Watchpoint Register Pairs* (WRPs) implemented. The meanings of the values of this field are:

0b0000 1 WRP implemented
0b0001 2 WRPs implemented
0b0010 3 WRPs implemented
. .
. .
. .
0b1111 16 WRPs implemented.

The minimum number of WRPs is 1.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.2 デバッグ識別レジスタ

このセクションは、次のサブセクションから構成されています。

- *デバッグ ID レジスタ (DBGDIDR)*
- *デバッグデバイス ID レジスタ (DBGDEVID)* P. C10-6
- *デバッグ ROM アドレスレジスタ (DBGDRAR)* P. C10-7
- *デバッグ自己アドレスオフセット レジスタ (DBGDSAR)* P. C10-8

C10.2.1 デバッグ ID レジスタ (DBGDIDR)

デバッグ ID レジスタ (DBGDIDR) は、次の内容を指定します。

- 実装されているデバッグアーキテクチャのバージョン
- デバッグ実装の一部の機能

DBGDIDR の特徴は次のとおりです。

- デバッグレジスタ 0、オフセット `0x000`
- 読み出し専用レジスタ
- v6 デバッグおよびそれ以降のすべてのバージョンのデバッグアーキテクチャに必須
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

DBGDIDR のフォーマットは次のとおりです。

WRP、ビット [31:28]

実装されている *ウォッチポイント レジスタペア* (WRP) の数。このフィールドの値は、次の意味を持ちます。

0b0000 実装されている WRP は 1 つです。
0b0001 実装されている WRP は 2 つです。
0b0010 実装されている WRP は 3 つです。
. .
. .
. .
0b1111 実装されている WRP は 16 個です。

WRP の最小数は 1 です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

BRPs, bits [27:24]

The number of *Breakpoint Register Pairs* (BRPs) implemented. The meanings of the values of this field are:

0b0000 Reserved

0b0001 2 BRPs implemented

0b0010 3 BRPs implemented

. .

. .

. .

0b1111 16 BRPs implemented.

The minimum number of BRPs is 2.

CTX_CMPs, bits [23:20]

The number of BRPs that can be used for Context ID comparison. The meanings of the values of this field are:

0b0000 1 BRP can be used for Context ID comparison

0b0001 2 BRPs can be used for Context ID comparison

0b0010 3 BRPs can be used for Context ID comparison

. .

. .

. .

0b1111 16 BRPs can be used for Context ID comparison.

The minimum number of BRPs with Context ID comparison capability is 1. The value in this field cannot be greater than the value in the BRPs field, bits [27:24].

The breakpoint comparators with Context ID comparison capability *must* be the highest addressed comparators. For example, if six comparators are implemented and two have Context ID comparison capability, the comparators with Context ID comparison capability must be comparators 4 and 5.

Version, bits [19:16]

The Debug architecture version. The permitted values of this field are:

0b0001 ARMv6, v6 Debug architecture

0b0010 ARMv6, v6.1 Debug architecture

0b0011 ARMv7 Debug architecture - Extended CP14 interface implemented

0b0100 ARMv7 Debug architecture - No Extended CP14 interface implemented.

All other values are reserved.

DEVID_imp, bit [15]

Debug Device ID Register, DBGDEVID, implemented bit. The meanings of the values of this bit are:

0 DBGDEVID is not implemented. Debug register 1010 is reserved.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

BRP、ビット [27:24]

実装されている *ブレークポイント レジスタペア*(BRP) の数。このフィールドの値は、次の意味を持ちます。

0b0000 予約

0b0001 実装されている BRP は 2 つです。

0b0010 実装されている BRP は 3 つです。

. .

. .

. .

0b1111 実装されている BRP は 16 個です。

BRP の最小数は 2 です。

CTX_CMP、ビット [23:20]

コンテキスト ID 比較に使用できる BRP の数。このフィールドの値は、次の意味を持ちます。

0b0000 コンテキスト ID 比較に使用できる BRP は 1 つです。

0b0001 コンテキスト ID 比較に使用できる BRP は 2 つです。

0b0010 コンテキスト ID 比較に使用できる BRP は 3 つです。

. .

. .

. .

0b1111 コンテキスト ID 比較に使用できる BRP は 16 個です。

コンテキスト ID 比較機能を持つ BRP の最小数は 1 です。このフィールドの値は、BRP フィールド（ビット [27:24]）の値を超えることはできません。

コンテキスト ID 比較機能を持つブレークポイントコンパレータは、最も高いアドレスに存在するコンパレータの*必要があります*。例えば、6 つのコンパレータが実装されており、そのうち 2 つがコンテキスト ID 比較機能を持っている場合、コンテキスト ID 比較機能を持つコンパレータはコンパレータ 4 と 5 であることが必要です。

バージョン、ビット [19:16]

デバッグアーキテクチャのバージョン。このフィールドに許容される値は次のとおりです。

0b0001 ARMv6、v6 デバッグアーキテクチャ

0b0010 ARMv6、v6.1 デバッグアーキテクチャ

0b0011 ARMv7 デバッグアーキテクチャで、拡張 CP14 インタフェースが実装されている。

0b0100 ARMv7 デバッグアーキテクチャで、拡張 CP14 インタフェースが実装されていない。

他の値はすべて予約されています。

DEVID_imp、ビット [15]

デバッグデバイス ID レジスタ (DBGDEVID) が実装されているかどうかを示すビット。このビットの値は、次の意味を持ちます。

0 DBGDEVID が実装されていません。デバッグレジスタ 1010 は予約です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

1 DBGDEVID is implemented, see *Debug Device ID Register (DBGDEVID)* on page C10-6.

This bit is always RAZ in ARMv6.

nSUHD_imp, bit [14]

Secure User halting debug not implemented bit. When the Security Extensions are implemented, the meanings of the values of this bit are:

0 Secure User halting debug is implemented

1 Secure User halting debug is not implemented.

If the Security Extensions are not implemented:

- Secure User halting debug cannot be implemented
- this bit is RAZ.

A v6.1 Debug processor that implements the Security Extensions must support Secure User halting debug. In v6.1 Debug this bit is always RAZ.

See also Chapter C2 *Invasive Debug Authentication*.

PCSR_imp, bit [13]

Program Counter Sampling Register (DBGPCSR) implemented as register 33 bit. The meanings of the values of this bit are:

0 DBGPCSR is not implemented as register 33

1 DBGPCSR is implemented as register 33.

Note

In v7 Debug, the DBGPCSR can be implemented as register 33, as register 40, or as both register 33 and register 40, as described in *Implemented Program Counter sampling registers* on page C8-2. The PCSR_imp bit only indicates whether it is implemented as register 33. For details of how to determine whether it is implemented as register 40 see *Debug Device ID Register (DBGDEVID)* on page C10-6.

In ARMv6, the Program Counter Sampling Register is an IMPLEMENTATION DEFINED feature of the external debug interface and is not indicated in the DBGDIDR. This bit is always RAZ in ARMv6.

See also *Program Counter Sampling Register (DBGPCSR)* on page C10-38.

SE_imp, bit [12]

Security Extensions implemented bit. The meanings of the values of this bit are:

0 Security Extensions are not implemented

1 Security Extensions are implemented.

v6 Debug is not a permitted option for an implementation that includes the Security Extensions. This bit is RAZ on a v6 Debug implementation.

Bits [11:8] Reserved, RAZ.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

1 DBGDEVID が実装されています。*デバッグデバイス ID レジスタ (DBGDEVID)* P. C10-6 を参照して下さい。

ARMv6 では、このビットは常に RAZ です。

nSUHD_imp、ビット [14]

セキュアユーザ ホールトデバッグの非実装を示すビット。セキュリティ拡張機能が実装されている場合、このビットの値は次の意味を持ちます。

0 セキュアユーザ ホールトデバッグが実装されています。

1 セキュアユーザ ホールトデバッグが実装されていません。

セキュリティ拡張機能が実装されていない場合、次の条件が適用されます。

- セキュアユーザ ホールトデバッグは実装できません。
- このビットは RAZ です。

セキュリティ拡張機能を実装している v6.1 デバッグプロセッサでは、セキュアユーザ ホールトデバッグをサポートする必要があります。v6.1 デバッグでは、このビットは常に RAZ です。

第 C2 章*「侵襲性デバッグの認証」*も参照して下さい。

PCSR_imp、ビット [13]

プログラムカウンタサンプリング レジスタ (DBGPCSR) がレジスタ 33 として実装されていることを示すビット。このビットの値は、次の意味を持ちます。

0 DBGPCSR がレジスタ 33 として実装されていません。

1 DBGPCSR がレジスタ 33 として実装されています。

注

v7 デバッグでは、*実装されているプログラムカウンタサンプリング レジスタ* P. C8-2 に説明されているように、DBGPCSR はレジスタ 33、レジスタ 40、または 33 と 40 の両方として実装可能です。PCSR_imp ビットは、レジスタ 33 として実装されているかどうかのみを示します。レジスタ 40 として実装されているかどうかを判定する方法の詳細については、*デバッグデバイス ID レジスタ (DBGDEVID)* P. C10-6 を参照して下さい。

ARMv6 では、プログラムカウンタサンプリング レジスタは、外部デバッグインタフェースの実装定義の機能で、DBGDIDR では示されません。ARMv6 では、このビットは常に RAZ です。

プログラムカウンタサンプリング レジスタ (DBGPCSR) P. C10-38 も参照して下さい。

SE_imp、ビット [12]

セキュリティ拡張機能の実装ビット。このビットの値は、次の意味を持ちます。

0 セキュリティ拡張機能が実装されていません。

1 セキュリティ拡張機能が実装されています。

v6 デバッグは、セキュリティ拡張機能を含む実装に許可されたオプションではありません。v6 デバッグ実装では、このビットは RAZ です。

ビット [11:8] 予約、RAZ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Variant, bits [7:4]

This field holds an IMPLEMENTATION DEFINED variant number. This number is incremented on functional changes. The value must match bits [23:20] of the CP15 Main ID Register.

Revision, bits [3:0]

This field holds an IMPLEMENTATION DEFINED revision number. This number is incremented on functional changes. The value must match bits [3:0] of the CP15 Main ID Register.

For details of the CP15 Main ID Register see:

- *c0, Main ID Register (MIDR)* on page B3-81, for a VMSA implementation
- *c0, Main ID Register (MIDR)* on page B4-32, for a PMSA implementation.

C10.2.2 Debug Device ID Register (DBGDEVID)

The Debug Device ID Register, DBGDEVID, extends the DBGDIDR by describing other features of the debug implementation.

The DBGDEVID register is:

- debug register 1010, at offset `0xFC8`
- a read-only register
- an optional register, that can be implemented only in v7 Debug
- when the Security Extensions are implemented, a Common register.

The DBGDIDR.DEVID_imp bit indicates whether the DBGDEVID register is implemented, see *Debug ID Register (DBGDIDR)* on page C10-3.

If the DBGDEVID register is not implemented:

- the Program Counter Sampling Register (DBGPCSR) is not implemented as register 40
- the Context ID Sampling Register (DBGCIDSR) is not implemented.

The format of the DBGDEVID register is:

31 … 4	3 … 0
Reserved, RAZ	PCsample

Bits [31:4] Reserved, RAZ.

PCsample, bits [3:0]

This field indicates the level of Program Counter sampling support using debug registers 40 and 41. The permitted values of this field are:

0b0000 Program Counter Sampling Register (DBGPCSR) is not implemented as register 40, and Context ID Sampling Register (DBGCIDSR) is not implemented.

0b0001 DBGPCSR is implemented as register 40, and DBGCIDSR is not implemented.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

バリアント、ビット [7:4]

このフィールドには、実装定義のバリアント番号が保持されています。この番号は、機能の変更時にインクリメントされます。この値は、CP15 メイン ID レジスタのビット [23:20] と一致している必要があります。

リビジョン、ビット [3:0]

このフィールドには、実装定義のリビジョン番号が保持されています。この番号は、機能の変更時にインクリメントされます。この値は、CP15 メイン ID レジスタのビット [3:0] と一致している必要があります。

CP15 メイン ID レジスタの詳細については、次のセクションを参照して下さい。

- *c0、メイン ID レジスタ (MIDR)* P. B3-81（VMSA 実装の場合）
- *c0、メイン ID レジスタ (MIDR)* P. B4-32（PMSA 実装の場合）

C10.2.2 デバッグデバイス ID レジスタ (DBGDEVID)

デバッグデバイス ID レジスタ (DBGDEVID) は、デバッグ実装の他の機能についての説明を加えることにより、DBGDIDR を拡張するものです。

DBGDEVID レジスタには、次の特徴があります。

- デバッグレジスタ 1010、オフセット `0xFC8`
- 読み出し専用レジスタ
- v7 デバッグにのみ実装可能なオプションのレジスタ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

DBGDIDR.DEVID_imp ビットは、DBGDEVID レジスタが実装されているかどうきを示します。*デバッグ ID レジスタ (DBGDIDR)* P. C10-3 を参照して下さい。

DBGDEVID レジスタが実装されていない場合、次の条件が適用されます。

- プログラムカウンタサンプリング レジスタ (DBGPCSR) はレジスタ 40 として実装されていません。
- コンテキスト ID サンプリングレジスタ (DBGCIDSR) は実装されていません。

DBGDEVID レジスタのフォーマットは次のとおりです。

31 … 4	3 … 0
予約、RAZ	PCsample

ビット [31:4] 予約、RAZ

PCsample、ビット [3:0]

このフィールドは、デバッグレジスタ 40 および 41 を使用するプログラムカウンタ サンプリングのサポートレベルを示します。このフィールドに許容される値は次のとおりです。

0b0000 プログラムカウンタサンプリング レジスタ (DBGPCSR) はレジスタ 40 として実装されておらず、コンテキスト ID サンプリングレジスタ (DBGCIDSR) は実装されていません。

0b0001 DBGPCSR はレジスタ 40 として実装されており、DBGCIDSR は実装されていません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b0010 DBGPCSR is implemented as register 40, and DBGCIDSR is implemented as register 41.

Other values are reserved.

Note

The DBGPCSR can be implemented as register 33, as register 40, or as both register 33 and register 40, as described in *Implemented Program Counter sampling registers* on page C8-2. The PCsample field only indicates whether it is implemented as register 40. For details of how to determine whether it is implemented as register 33 see *Debug ID Register (DBGDIDR)* on page C10-3.

C10.2.3 Debug ROM Address Register (DBGDRAR)

The Debug ROM Address Register, DBGDRAR, defines the base address of a ROM Table, that locates and describes the debug components in the system.

The DBGDRAR is:

- Only implemented through the Baseline CP14 interface, and therefore does not have a register number and offset. For more information, see *The CP14 debug register interfaces* on page C6-32.
- A read-only register.
- Implemented as follows:

  ARMv6 This register is not defined in ARMv6.

  v7 Debug If no Memory-mapped debug components, including this processor, are implemented, this register is RAZ.

  Otherwise, the register defines the physical address in memory of a ROM Table.
- When the Security Extensions are implemented, a Common register.

It is IMPLEMENTATION DEFINED how the processor determines the value that is returned as the ROM Table address. If the processor cannot determine the value, the Valid field in the register must be RAZ.

One implementation scheme is to provide inputs **DBGROMADDR[31:12]** and **DBGROMADDRV** that a system designer must tie-off to the correct value. **DBGROMADDRV** must be tied HIGH only if **DBGROMADDR[31:12]** is tied off to a valid value, otherwise **DBGROMADDR[31:12]** and **DBGROMADDRV** must be tied LOW.

The format of the DBGDRAR is:

31 … 12	11 … 2	1	0
DBGROMADDR[31:12]	Reserved, RAZ		Valid

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b0010 DBGPCSR はレジスタ 40 として実装されており、DBGCIDSR はレジスタ 41 として実装されています。

その他の値はすべて予約です。

注

実装されているプログラムカウンタサンプリング レジスタ P. C8-2 に説明されているように、DBGPCSR はレジスタ 33、レジスタ 40、または 33 と 40 の両方として実装可能です。PCsample フィールドは、レジスタ 40 として実装されているかどうかのみを示します。レジスタ 33 として実装されているかどうかを判定する方法の詳細については、*デバッグ ID レジスタ (DBGDIDR)* P. C10-3 を参照して下さい。

C10.2.3 デバッグ ROM アドレスレジスタ (DBGDRAR)

デバッグ ROM アドレスレジスタ (DBGDRAR) は、システムのデバッグコンポーネントの位置と記述を示している ROM テーブルのベースアドレスを定義します。

DBGDRAR の特徴は次のとおりです。

- ベースライン CP14 インタフェース経由でのみ実装されるため、レジスタ番号やオフセットは存在しません。詳細については、*CP14 デバッグレジスタ インタフェース* P. C6-32 を参照して下さい。
- 読み出し専用レジスタ
- 次のように実装されます。

  ARMv6 ARMv6 ではこのレジスタは定義されていません。

  v7デバッグ メモリマップされたデバッグコンポーネントが、このプロセッサを含めて何も実装されていない場合、このレジスタは RAZ です。

  それ以外の場合、このレジスタは ROM テーブルのメモリ内の物理アドレスを定義します。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

ROM テーブルアドレスとして返される値をプロセッサがどのように決定するかは、実装定義です。プロセッサが値を決定できない場合、レジスタの有効フィールドは RAZ の必要があります。

実装手法の 1 つは、DBGROMADDR[31:12] および DBGROMADDRV 入力を提供することです。システム設計者は、これらの入力を正しい値に固定する必要があります。DBGROMADDRV は、DBGROMADDR[31:12] が有効な値に固定されている場合のみ、HIGH に固定する必要があります。それ以外の場合、DBGROMADDR[31:12] と DBGROMADDRV は LOW に固定する必要があります。

DBGDRAR のフォーマットは次のとおりです。

31 … 12	11 … 2	1	0
DBGROMADDR[31:12]	予約、RAZ		有効

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

This register format applies regardless of the implementation scheme for identifying the ROM Table address.

DBGROMADDR[31:12], bits [31:12]

Bits [31:12] of the ROM Table physical address. Bits [11:0] of the address are zero.

If the Valid field, bits [1:0], is zero the value of this field is UNKNOWN.

Bits [11:2] Reserved, RAZ.

Valid, bits [1:0]

This field indicates whether the ROM Table address is valid. In the recommended implementation it reflects the value of the **DBGROMADDRV** signal, and the permitted values of this field are:

0b00 **DBGROMADDRV** is LOW, ROM Table address is not valid

0b11 **DBGROMADDRV** is HIGH, ROM Table address is valid.

Other values are reserved.

The ROM Table contains a zero-terminated list of signed 32-bit offsets from the ROM Table base to other Memory-mapped debug components in the system. All the debug components pointed to must contain a set of debug component identification registers compatible with the format in *Debug Component Identification Registers (DBGCID0 to DBGCID3)* on page C10-102. For more information, see the *ARM Debug Interface v5 Architecture Specification*.

C10.2.4 Debug Self Address Offset Register (DBGDSAR)

The Debug Self Address Offset Register, DBGDSAR, defines the offset from the ROM Table physical address to the physical address of the debug registers for the processor.

The DBGDSAR is:

- Only implemented through the Baseline CP14 interface, and therefore does not have a register number and offset. For more information, see *The CP14 debug register interfaces* on page C6-32.
- A read-only register.
- Implemented as follows:

  ARMv6 This register is not defined in ARMv6.

  v7 Debug If no memory-mapped interface is provided, this register is RAZ.

  Otherwise, the register gives the offset from the ROM Table physical address to the physical address of the debug registers for the processor.
- When the Security Extensions are implemented, a Common register.

It is IMPLEMENTATION DEFINED how the processor determines the value that is returned as the debug self address offset. If the processor cannot determine the value, the Valid field in the register must be RAZ.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

このレジスタフォーマットは、ROM テーブルアドレスを識別するための実装手法にかかわらず適用されます。

DBGROMADDR[31:12]、ビット [31:12]

ROM テーブルのテーブル物理アドレスのビット [31:12]。アドレスのビット [11:0] は 0 です。

有効フィールド（ビット [1:0]）が 0 の場合、このフィールドの値は未知です。

ビット [11:2] 予約、RAZ

有効、ビット [1:0]

このフィールドは、ROM テーブルのアドレスが有効かどうかを示します。推奨される実装では、この値は DBGROMADDRV 信号の値を反映します。このフィールドに許可される値は次のとおりです。

0b00 DBGROMADDRV が LOW で、ROM テーブルのアドレスは無効です。

0b11 DBGROMADDRV が HIGH で、ROM テーブルのアドレスは有効です。

その他の値はすべて予約です。

ROM テーブルには、ROM テーブルベースから、システムの他のメモリマップされたデバッグコンポーネントへの符号付き 32 ビットオフセットが含まれており、その一覧の最後は 0 です。指し示されるデバッグコンポーネントにはすべて、*デバッグコンポーネント識別レジスタ (DBGCID0 ～ DBGCID3)* P. C10-102 で示されているフォーマットと互換の、デバッグコンポーネント識別レジスタの組が含まれている必要があります。詳細については、『*ARM デバッグインタフェース v5、アーキテクチャ仕様書*』を参照して下さい。

C10.2.4 デバッグ自己アドレスオフセット レジスタ (DBGDSAR)

デバッグ自己アドレスオフセット レジスタ (DBGDSAR) は、ROM テーブルの物理アドレスから、プロセッサのデバッグレジスタの物理アドレスへのオフセットを定義します。

DBGDSAR の特徴は次のとおりです。

- ベースライン CP14 インタフェース経由でのみ実装されるため、レジスタ番号やオフセットは存在しません。詳細については、*CP14 デバッグレジスタ インタフェース* P. C6-32 を参照して下さい。
- 読み出し専用レジスタ
- 次のように実装されます。

  ARMv6 ARMv6 ではこのレジスタは定義されていません。

  v7デバッグ メモリマップされたインタフェースが提供されない場合、このレジスタは RAZ です。

  それ以外の場合、このレジスタは、ROM テーブルの物理アドレスから、プロセッサのデバッグレジスタの物理アドレスへのオフセットを定義します。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

デバッグ自己アドレスオフセットとして返される値をプロセッサがどのように決定するかは、実装定義です。プロセッサが値を決定できない場合、レジスタの有効フィールドは RAZ の必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

One implementation scheme is to provide inputs **DBGSELFADDR[31:12]** and **DBGSELFADDRV** that a system designer must tie-off to the correct value. **DBGSELFADDRV** must be tied HIGH only if **DBGSELFADDR[31:12]** is tied off to a valid value, otherwise **DBGSELFADDR[31:12]** and **DBGSELFADDRV** must be tied LOW.

The format of the DBGDSAR is:

31 … 12	11 … 2	1 0
DBGSELFADDR[31:12]	Reserved, RAZ	Valid

This register format applies regardless of the implementation scheme for identifying the debug self address offset.

DBGSELFADDR [31:12], bits [31:12]

Bits [31:12] of the two's complement offset from the ROM Table physical address to the physical address where the debug registers are mapped. Bits [11:0] of the address are zero.

If the Valid field, bits [1:0], is zero the value of this field is UNKNOWN.

Bits [11:2] Reserved, RAZ.

Valid, bits [1:0]

This field indicates whether the debug self address offset is valid. In the recommended implementation it reflects the value of the **DBGSELFADDRV** signal, and the permitted values of this field are:

0b00 **DBGSELFADDRV** is LOW, offset is not valid

0b11 **DBGSELFADDRV** is HIGH, offset is valid.

Other values are reserved.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

実装手法の 1 つは、DBGSELFADDR[31:12] および DBGSELFADDRV 入力を提供することです。システム設計者は、これらの入力を正しい値に固定する必要があります。DBGSELFADDRV は、DBGSELFADDR[31:12] が有効な値に固定されている場合のみ、HIGH に固定する必要があります。それ以外の場合、DBGSELFADDR[31:12] と DBGSELFADDRV は LOW に固定する必要があります。

DBGDSAR のフォーマットは次のとおりです。

31 … 12	11 … 2	1 0
DBGSELFADDR[31:12]	予約、RAZ	有効

このレジスタフォーマットは、デバッグ自己アドレスオフセットを識別するための実装手法にかかわらず適用されます。

DBGSELFADDR[31:12]、ビット [31:12]

ROM テーブルの物理アドレスから、デバッグレジスタがマッピングされている物理アドレスへの、2 の補数オフセットのビット [31:12] を指定します。アドレスのビット [11:0] は 0 です。

有効フィールド（ビット [1:0]）が 0 の場合、このフィールドの値は未知です。

ビット [11:2] 予約、RAZ

有効、ビット [1:0]

このフィールドは、デバッグ自己アドレスオフセットが有効かどうかを示します。推奨される実装では、この値は DBGSELFADDRV 信号の値を反映します。このフィールドに許可される値は次のとおりです。

0b00 DBGSELFADDRV が LOW で、オフセットは無効です。

0b11 DBGSELFADDRV が HIGH で、オフセットは有効です。

その他の値はすべて予約です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.3 Control and status registers

This section contains the following subsections:

- *Debug Status and Control Register (DBGDSCR)*
- *Watchpoint Fault Address Register (DBGWFAR)* on page C10-28
- *Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29
- *Device Power-down and Reset Control Register (DBGPRCR), v7 Debug only* on page C10-31
- *Device Power-down and Reset Status Register (DBGPRSR), v7 Debug only* on page C10-34
- *Program Counter Sampling Register (DBGPCSR)* on page C10-38
- *Context ID Sampling Register (DBGCIDSR)* on page C10-39.

C10.3.1 Debug Status and Control Register (DBGDSCR)

The Debug Status and Control Register, DBGDSCR, provides the main control register for the debug facilities in the ARM architecture. All debug implementations provide both internal and external views of the DBGDSCR, and it is the external view that provides control of the debug facilities. These views are referred to as:

DBGDSCRint the internal view

DBGDSCRext the external view.

For more information, see *Internal and external views of the DBGDSCR and the DCC registers* on page C6-21.

The DBGDSCR:

- In its external view:
  - — is debug register 34 at offset `0x088`.
  - — is a read/write register, with more restricted access to some bits.
- Has the following differences in different versions of the Debug architecture:
  - — v6.1 Debug and v7 Debug define additional bits in the register.
  - — DBGDSCR bit [9] is defined only in v6 Debug and v6.1 Debug. It is reserved in v7 Debug.
  - — Access to the register depends on the version of the Debug architecture, see *Access to the DBGDSCR* on page C10-27.
  - — The behavior of RXfull and TXfull on reads of DBGDSCR through the internal and external views is different, see *Access controls on the external view of the DCC registers and DBGITR, v7 Debug only* on page C10-21.
- When the Security Extensions are implemented, is a Common register.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C10.3 制御およびステータスレジスタ

このセクションは、次のサブセクションから構成されています。

- *デバッグステータスおよび制御レジスタ (DBGDSCR)*
- *ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10- 28
- *デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10- 29
- *デバイス電力オフおよびリセット制御レジスタ (DBGPRCR)、v7 デバッグのみ* P. C10- 31
- *デバイス電力オフおよびリセットステータス レジスタ (DBGPRSR)、v7 デバッグのみ* P. C10- 34
- *プログラムカウンタサンプリング レジスタ (DBGPCSR)* P. C10- 38
- *コンテキスト ID サンプリングレジスタ (DBGCIDSR)* P. C10- 39

C10.3.1 デバッグステータスおよび制御レジスタ (DBGDSCR)

デバッグステータスおよび制御レジスタ (DBGDSCR) は、ARM アーキテクチャのデバッグ機能のメイン制御レジスタを提供します。すべてのデバッグ実装では、DBGDSCR の内部ビューと外部ビューの両方を提供し、デバッグ機能の制御は外部ビューによって提供されます。これらのビューは、次の名前で呼ばれます。

DBGDSCRint 内部ビュー

DBGDSCRext 外部ビュー

詳細については、*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6- 21 を参照して下さい。

DBGDSCR には次の特徴があります。

- 外部ビューには次の特徴があります。
  - – デバッグレジスタ 34、オフセット `0x088`
  - – 読み出し / 書き込みレジスタで、一部のビットはアクセスがさらに制限されています。
- デバッグアーキテクチャのバージョンにより、次の相違点があります。
  - – v6.1 デバッグと v7 デバッグでは、レジスタに追加のビットが定義されています。
  - – DBGDSCR ビット [9] は、v6 デバッグと v6.1 デバッグでのみ定義されています。v7 デバッグでは、このビットは予約されています。
  - – レジスタへのアクセスは、デバッグアーキテクチャのバージョンによって異なります。*DBGDSCR へのアクセス* P. C10- 27 を参照して下さい。
  - – DBGDSCR を読み出すときの RXfull と TXfull の動作は、内部ビューと外部ビューのどちらを経由して読み出すかによって異なります。*DCC レジスタと DBGITR の外部ビューへのアクセス制御、v7 デバッグのみ* P. C10- 21 を参照して下さい。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In v7 Debug, the format of the DBGDSCR is:

Bits [31,28,23:22]

Reserved, UNK/SBZP.

RXfull, bit [30]

The DBGDTRRX Register full bit. The possible values of this bit are:

0 DBGDTRRX Register empty

1 DBGDTRRX Register full.

Normally, RXfull is:

- set to 1 on writes to DBGDTRRXext
- cleared to 0 on reads of DBGDTRRXint.

For more information about the behavior of RXfull and the DBGDTRRX Register see *Host to Target Data Transfer Register (DBGDTRRX)* on page C10-40.

TXfull, bit [29]

The DBGDTRTX Register full bit. The possible values of this bit are:

0 DBGDTRTX Register empty

1 DBGDTRTX Register full.

Normally, TXfull is:

- cleared to 0 on reads of DBGDTRTXext
- set to 1 on writes to DBGDTRTXint.

For more information about the behavior of TXfull and the DBGDTRTX Register see *Target to Host Data Transfer Register (DBGDTRTX)* on page C10-43.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

v7 デバッグでは、DBGDSCR のフォーマットは次のとおりです。

ビット [31、28、23:22]

予約、UNK/SBZP

RXfull、ビット [30]

DBGDTRRX レジスタフル ビット。このビットの可能な値は次のとおりです。

0 DBGDTRRX レジスタが空です。

1 DBGDTRRX レジスタがいっぱいです。

通常、RXfull は次のように動作します。

- DBGDTRRXext への書き込み時に 1 にセットされます。
- DBGDTRRXint の読み出し時に 0 にクリアされます。

RXfull および DBGDTRRX レジスタの動作の詳細については、*ホストからターゲットへのデータ転送レジスタ (DBGDTRRX)* P. C10- 40 を参照して下さい。

TXfull、ビット [29]

DBGDTRTX レジスタフル ビット。このビットの可能な値は次のとおりです。

0 DBGDTRTX レジスタが空です。

1 DBGDTRTX レジスタがいっぱいです。

通常、TXfull は次のように動作します。

- DBGDTRTXext の読み出し時に 0 にクリアされます。
- DBGDTRTXint への書き込み時に 1 にセットされます。

TXfull および DBGDTRTX レジスタの動作の詳細については、*ターゲットからホストへのデータ転送レジスタ (DBGDTRTX)* P. C10- 43 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

RXfull_l, bit [27], v7 Debug only

The latched RXfull bit. This is a copy of the RXfull bit taken as a side-effect of a read of DBGDSCRext. This means that RXfull_l holds the last value of RXfull read from DBGDSCRext:

- On a read of DBGDSCRext that is permitted to have side-effects, the value of RXfull_l is the same as the value of RXfull.
- On a read of DBGDSCRint, the value of RXfull_l is UNKNOWN.

Normally, RXfull_l is set to 1 on writes to DBGDTRRXext.

The RXfull_l bit controls the behavior of the processor on writes to DBGDTRRXext. For more information about the behavior of RXfull_l and the DBGDTRRX Register, see *Host to Target Data Transfer Register (DBGDTRRX)* on page C10-40.

TXfull_l, bit [26], v7 Debug only

The latched TXfull bit. This is a copy of the TXfull bit taken as a side-effect of a read of DBGDSCRext. This means that TXfull_l holds the last value of TXfull read from DBGDSCRext:

- On a read of DBGDSCRext that is permitted to have side-effects, the value of TXfull_l is the same as the value of TXfull.
- On a read of DBGDSCRint, the value of TXfull_l is UNKNOWN.

Normally, TXfull_l is cleared to 0 on reads of DBGDTRTXext.

The TXfull_l bit controls the behavior of the processor on reads of DBGDTRTXext. For more information about the behavior of TXfull_l and the DBGDTRTX Register see *Target to Host Data Transfer Register (DBGDTRTX)* on page C10-43.

PipeAdv, bit [25], v7 Debug only

Sticky Pipeline Advance bit. This bit is set to 1 every time the processor pipeline retires one instruction. It is cleared to 0 by a write to DBGDRCR[3], see *Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29.

This flag enables a debugger to detect that the processor is idle. In some situations this might indicate that the processor is deadlocked.

InstrCompl_l, bit [24], v7 Debug only

The latched Instruction Complete bit. This is a copy of the Instruction Complete internal flag, taken on each read of DBGDSCRext. InstrCompl signals whether the processor has completed execution of an instruction issued through the Instruction Transfer Register (DBGITR), see *Instruction Transfer Register (DBGITR)* on page C10-46. InstrCompl is not visible directly in any register.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

RXfull_l、ビット [27]、v7 デバッグのみ

ラッチされた RXfull ビット。このビットは、DBGDSCRext の読み出しの副作用として取得される RXfull ビットのコピーです。すなわち、RXfull_l は、DBGDSCRexe から最後に読み出された RXfull の値を保持します。

- DBGDSCRext の読み出しで副作用が許容される場合、RXfull_l の値は RXfull の値と同じです。
- DBGDSCRint の読み出し時は、RXfull_l の値は未知です。

通常、RXfull_l は DBGDTRRXext への書き込み時に 1 にセットされます。

RXfull_l ビットは、DBGDTRRXext への書き込み時におけるプロセッサの動作を制御します。RXfull_l および DBGDTRRX レジスタの動作の詳細については、*ホストからターゲットへのデータ転送レジスタ (DBGDTRRX)* P. C10- 40 を参照して下さい。

TXfull_l、ビット [26]、v7 デバッグのみ

ラッチされた TXfull ビット。このビットは、DBGDSCRext の読み出しの副作用として取得される TXfull ビットのコピーです。すなわち、TXfull_l は、DBGDSCRexe から最後に読み出された TXfull の値を保持します。

- DBGDSCRext の読み出しで副作用が許容される場合、TXfull_l の値は TXfull の値と同じです。
- DBGDSCRint の読み出し時は、TXfull_l の値は未知です。

通常、TXfull_l は DBGDTRTXext の読み出し時に 0 にクリアされます。

TXfull_l ビットは、DBGDTRTXext の読み出し時におけるプロセッサの動作を制御します。TXfull_l および DBGDTRTX レジスタの動作の詳細については、*ターゲットからホストへのデータ転送レジスタ (DBGDTRTX)* P. C10- 43 を参照して下さい。

PipeAdv、ビット [25]、v7 デバッグのみ

スティッキーパイプライン アドバンスビット。このビットは、プロセッサのパイプラインが命令を終了するごとに 1 にセットされます。DBGDRCR[3] への書き込み時に 0 にクリアされます。*デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10- 29 を参照して下さい。

このフラグにより、デバッガはプロセッサがアイドルであることを検出できます。一部の状況では、これはプロセッサがデッドロックしていることを示す場合もあります。

InstrCompl_l、ビット [24]、v7 デバッグのみ

ラッチされた命令完了ビット。このビットは、DBGDSCRext の読み出しごとに取得される、命令完了内部フラグのコピーです。InstrCompl は、命令転送レジスタ (DBGITR) 経由で発行された命令の実行をプロセッサが完了したかどうかを通知します。*命令転送レジスタ (DBGITR)* P. C10- 46 を参照して下さい。InstrCompl は、どのレジスタでも直接可視ではありません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

On a read of DBGDSCRext, InstrCompl_l always returns the current value of InstrCompl. The meanings of the values of InstrCompl_l are:

0 an instruction previously issued through the DBGITR has not completed its changes to the architectural state of the processor

1 all instructions previously issued through the DBGITR have completed their changes to the architectural state of the processor.

Normally, InstrCompl:

- is cleared to 0 following issue of an instruction through DBGITR
- becomes 1 once the instruction completes.

The taking of an exception marks the completion of the instruction. InstrCompl is set to 1 if an instruction generates an Undefined Instruction or Data Abort exception.

InstrCompl is set to 1 on entry to Debug state. For more information about the behavior of InstrCompl, InstrCompl_l and the DBGITR, see:

- *Instruction Transfer Register (DBGITR)* on page C10-46
- *Host to Target Data Transfer Register (DBGDTRRX)* on page C10-40
- *Target to Host Data Transfer Register (DBGDTRTX)* on page C10-43.

Bits [27:24], v6 Debug and v6.1 Debug only

Reserved, UNK/SBZP.

ExtDCCmode, bits [21:20], v7 Debug only

The External DCC access mode field. This field controls the access mode for the external views of the DCC registers and the Instruction Transfer Register (DBGITR). Possible values are:

0b00 Non-blocking mode

0b01 Stall mode

0b10 Fast mode.

The values of 0b11 is reserved.

For details of the external DCC access modes see *Access controls on the external view of the DCC registers and DBGITR, v7 Debug only* on page C10-21.

Bits [21:20], v6 Debug and v6.1 Debug only

Reserved, UNK/SBZP.

ADAdiscard, bit [19], v6.1 Debug and v7 Debug

Asynchronous Data Aborts Discarded bit. The possible values of this bit are:

0 Asynchronous aborts handled normally

1 On an asynchronous abort, the processor sets the Sticky Asynchronous Data Abort bit, ADABORT_l, to 1 but otherwise discards the abort.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DBGDSCRext を読み出すと、InstrCompl_l は常に、InstrCompl の現在の値を返します。InstrCompl_l の値は、次の意味を持ちます。

0 以前に DBGITR 経由で発行された命令は、プロセッサのアーキテクチャ的な状態の変更を完了していません。

1 以前に DBGITR 経由で発行された命令のすべてが、プロセッサのアーキテクチャ的な状態の変更を完了しました。

通常、InstrCompl には次の規則が適用されます。

- DBGITR 経由で命令が発行された後に 0 にクリアされます。
- 命令が完了すると、1 になります。

例外の取得は、命令の完了をマークします。命令により未定義命令またはデータアボート例外が生成された場合、InstrCompl は 1 にセットされます。

デバッグ状態の開始時に、InstrCompl は 1 にセットされます。InstrCompl、InstrCompl_l、DBGITR の動作の詳細については、次のセクションを参照して下さい。

- *命令転送レジスタ (DBGITR)* P. C10- 46
- *ホストからターゲットへのデータ転送レジスタ (DBGDTRRX)* P. C10- 40
- *ターゲットからホストへのデータ転送レジスタ (DBGDTRTX)* P. C10- 43

ビット [27:24]、v6 デバッグおよび v6.1 デバッグのみ

予約、UNK/SBZP

ExtDCCmode、ビット [21:20]、v7 デバッグのみ

外部 DCC アクセスモード フィールド。このフィールドは、DCC レジスタと命令転送レジスタ (DBGITR) の外部ビューのアクセスモードを制御します。許される値は次のとおりです。

0b00 非ブロッキングモード

0b01 ストールモード

0b10 高速モード

値 0b11 は予約されています。

外部 DCC アクセスモードの詳細については、*DCC レジスタと DBGITR の外部ビューへのアクセス制御、v7 デバッグのみ* P. C10- 21 を参照して下さい。

ビット [21:20]、v6 デバッグおよび v6.1 デバッグのみ

予約、UNK/SBZP

ADAdiscard、ビット [19]、v6.1 デバッグおよび v7 デバッグのみ

非同期データアボートの破棄ビット。このビットの可能な値は次のとおりです。

0 非同期アボートは通常に処理されます。

1 非同期アボートが発生すると、プロセッサはスティッキー非同期データアボート ビット (ADABORT_l) を 1 にセットしますが、アボートを破棄します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

The conditions for setting ADABORT_l to 1 are different in v7 Debug and v6.1 Debug. For more information, see the description of the ADABORT_l bit, bit [7].

It is IMPLEMENTATION DEFINED whether the hardware automatically sets this bit to 1 on entry to Debug state, see *Asynchronous aborts and entry to Debug state* on page C5-5.

The processor clears this bit to 0 on exit from Debug state.

NS, bit [18], v6.1 Debug and v7 Debug

Non-secure state status bit. If the processor implements Security Extensions, this bit indicates whether the processor is in the Secure state. The possible values of this bit are:

0 the processor is in the Secure state

1 the processor is in the Non-secure state.

If the processor does not implement Security Extensions, this bit is RAZ.

SPNIDdis, bit [17], v6.1 Debug and v7 Debug

Secure Privileged Non-Invasive Debug Disabled bit. The behavior of this bit depends on the version of the Debug architecture:

v6.1 Debug

If the processor implements Security Extensions, this bit takes the value of the inverse of the **SPNIDEN** input. Otherwise it is RAZ.

v7 Debug This bit is the inverse of bit [6] of the Authentication Status Register, see *Authentication Status Register (DBGAUTHSTATUS)* on page C10-96.

SPIDdis, bit [16], v6.1 Debug and v7 Debug

Secure Privileged Invasive Debug Disabled bit. The behavior of this bit depends on the version of the Debug architecture:

v6.1 Debug

If the processor implements Security Extensions, this bit takes the value of the inverse of the **SPIDEN** input. Otherwise it is RAZ.

v7 Debug This bit is the inverse of bit [4] of the Authentication Status Register, see *Authentication Status Register (DBGAUTHSTATUS)* on page C10-96.

Bits [19:16], v6 Debug only

Reserved, UNK/SBZP.

MDBGen, bit [15]

Monitor debug-mode enable bit. The possible values of this bit are:

0 Monitor debug-mode disabled

1 Monitor debug-mode enabled.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

ADABORT_l を 1 にセットする条件は、v7 デバッグと v6.1 デバッグでは異なります。詳細については、ADABORT_l ビット（ビット [7]）の説明を参照して下さい。

デバッグ状態への遷移時に、ハードウェアがこのビットを自動的に 1 にセットするかどうかは実装定義です。*非同期アボートとデバッグ状態への遷移* P. C5-5 を参照して下さい。

プロセッサは、デバッグ状態の終了時にこのビットを 0 にクリアします。

NS、ビット [18]、v6.1 デバッグおよび v7 デバッグのみ

非セキュア状態ステータスビット。プロセッサにセキュリティ拡張機能が実装されている場合、このビットはプロセッサがセキュア状態であるかどうかを示します。このビットの可能な値は次のとおりです。

0 プロセッサはセキュア状態です。

1 プロセッサは非セキュア状態です。

プロセッサにセキュリティ拡張機能が実装されていない場合、このビットは RAZ です。

SPNIDdis、ビット [17]、v6.1 デバッグおよび v7 デバッグのみ

セキュアな特権非侵襲性デバッグのディセーブルビット。このビットの動作は、デバッグアーキテクチャのバージョンによって異なります。

v6.1 デバッグ

プロセッサにセキュリティ拡張機能が実装されている場合、このビットは SPNIDEN 入力の否定の値をとります。それ以外の場合は RAZ です。

v7デバッグ このビットは、認証ステータスレジスタのビット [6] の否定です。*認証ステータスレジスタ (DBGAUTHSTATUS)* P. C10-96 を参照して下さい。

SPIDdis、ビット [16]、v6.1 デバッグおよび v7 デバッグのみ

セキュアな特権侵襲性デバッグのディセーブルビット。このビットの動作は、デバッグアーキテクチャのバージョンによって異なります。

v6.1 デバッグ

プロセッサにセキュリティ拡張機能が実装されている場合、このビットは SPIDEN 入力の否定の値をとります。それ以外の場合は RAZ です。

v7デバッグ このビットは、認証ステータスレジスタのビット [4] の否定です。*認証ステータスレジスタ (DBGAUTHSTATUS)* P. C10-96 を参照して下さい。

ビット [19:16]、v6 デバッグのみ

予約、UNK/SBZP

MDBGen、ビット [15]

モニタデバッグモード イネーブルビット。このビットの可能な値は次のとおりです。

0 モニタデバッグモードは不可能です。

1 モニタデバッグモードは可能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If the external interface input **DBGEN** is LOW, the MDBGen bit reads as 0. The programmed value is masked until **DBGEN** is taken HIGH. When **DBGEN** goes HIGH, the value read and the behavior of the processor correspond to the programmed value.

Note

- If Halting debug-mode is enabled, by setting the HDBGen bit to 1, then the Monitor debug-mode setting is disabled regardless of the setting of the MDBGen bit.
- It is the programmed value of the MDBGen bit, not the value returned by reads of the DBGDSCR, that is saved by the OS Save and Restore Register in a power-down sequence. For more information, see *The OS Save and Restore mechanism* on page C6-8.

HDBGen, bit [14]

Halting debug-mode enable bit. The possible values of this bit are:

0 Halting debug-mode disabled

1 Halting debug-mode enabled.

If the external interface input **DBGEN** is LOW, the HDBGen bit reads as 0. The programmed value is masked until **DBGEN** is taken HIGH. When **DBGEN** goes HIGH, the value read and the behavior of the processor correspond to the programmed value.

Note

It is the programmed value of the HDBGen bit, not the value returned by reads of the DBGDSCR, that is saved by the OS Save and Restore Register in a power-down sequence. For more information, see *The OS Save and Restore mechanism* on page C6-8.

ITRen, bit [13]

Execute ARM instruction enable bit. This bit enables the execution of ARM instructions through the DBGITR, see *Instruction Transfer Register (DBGITR)* on page C10-46. The possible values of this bit are:

0 ITR mechanism disabled

1 The ITR mechanism for forcing the processor to execute instructions in Debug state via the external debug interface is enabled.

Setting this bit to 1 when the processor is in Non-debug state causes UNPREDICTABLE behavior. The effect of writing to DBGITR when this bit is set to 0 is UNPREDICTABLE.

The implementation of this bit can depend on the Debug architecture version:

ARMv6 If the external debug interface does not have a mechanism for forcing the processor to execute instructions in Debug state via the external debug interface, this bit is RAZ/WI.

v7 Debug This bit, and the DBGITR, are required.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

外部インタフェース入力 DBGEN が LOW の場合、MDBGen ビットは 0 として読み出されます。プログラムされた値は、DBGEN が HIGH になるまでマスクされます。DBGEN が HIGH になると、読み出される値とプロセッサの動作は、プログラムされた値に対応するようになります。

注

- HDBGen ビットが 1 にセットされ、ホールトデバッグ モードが可能なとき、MDBGen ビットの設定にかかわらず、モニタデバッグ モード設定は不可能です。
- 電力オフシーケンスで OS 保存および復元レジスタにによって保存されるのは、DBGDSCR の読み出しによって返される値ではなく、プログラムされた MDBGen ビットの値です。詳細については、*OS 保存および復元機構* P. C6-8 を参照して下さい。

HDBGen、ビット [14]

ホールトデバッグモード イネーブルビット。このビットの可能な値は次のとおりです。

0 ホールトデバッグ モードは不可能です。

1 ホールトデバッグ モードは可能です。

外部インタフェース入力 DBGEN が LOW の場合、HDBGen ビットは 0 として読み出されます。プログラムされた値は、DBGEN が HIGH になるまでマスクされます。DBGEN が HIGH になると、読み出される値とプロセッサの動作は、プログラムされた値に対応するようになります。

注

電力オフシーケンスで OS 保存および復元レジスタによって保存されるのは、DBGDSCR の読み出しによって返される値ではなく、プログラムされた HDBGen ビットの値です。詳細については、*OS 保存および復元機構* P. C6-8 を参照して下さい。

ITRen、ビット [13]

ARM 命令の実行イネーブルビット。このビットは、DBGITR 経由での ARM 命令の実行を可能にします。*命令転送レジスタ (DBGITR)* P. C10-46 を参照して下さい。このビットの可能な値は次のとおりです。

0 ITR 機構は非稼動です。

1 デバッグ状態で、外部デバッグインタフェース経由でプロセッサに強制的に命令を実行させる ITR 機構が稼動中です。

プロセッサが非デバッグ状態のときにこのビットを 1 にセットすると、予測不能な動作を引き起こします。このビットが 0 にセットされているときに DBGITR へ書き込みを行った場合、影響は予測不能です。

このビットの実装は、デバッグアーキテクチャのバージョンによって異なる可能性があります。

ARMv6 外部デバッグインタフェースに、デバッグ状態で外部デバッグインタフェースを経由して、プロセッサに強制的に命令を実行させる機構が搭載されていない場合、このビットは RAZ/WI です。

v7デバッグ このビットと DBGITR は必須です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

UDCCdis, bit [12]

User mode access to Communications Channel disable bit. The possible values of this bit are:

0 User mode access to Communication Channel enabled

1 User mode access to Communication Channel disabled.

When this bit is set to 1, if a User mode process tries to access the DBGDIDR, DBGDSCRint, DBGDTRRXint, or DBGDTRTXint through CP14 operations, the Undefined Instruction exception is taken. Setting this bit to 1 prevents User mode access to any CP14 debug register.

INTdis, bit [11]

Interrupts Disable bit. This bit can be used to mask the taking of IRQs and FIQs. The possible values of this bit are:

0 interrupts enabled

1 interrupts disabled.

If the external debugger needs to execute a piece of code in Non-debug state as part of the debugging process, but that code must not be interrupted, the external debugger sets this bit to 1.

For example, when single stepping code in a system with a periodic timer interrupt, the period of the interrupt is likely to be more frequent than the stepping frequency of the debugger. In this situation, if the debugger steps the target without setting the INTdis bit to 1 for the duration of the step, the interrupt is pending. This means that, if interrupts are enabled in the CPSR, the interrupt is taken as soon as the processor leaves Debug state.

The INTdis bit is ignored when either:

- DBGDSCR[15:14] == 0b00
- **DBGEN** is LOW.

For more information about the debug authentication signals see Chapter C2 *Invasive Debug Authentication*.

Note

If implemented, the ISR always reflects the status of the IRQ and FIQ signals, regardless of the value of the INTdis bit. For more information, see *c12, Interrupt Status Register (ISR)* on page B3-150.

DBGack, bit [10]

Force Debug Acknowledge bit. A debugger can use this bit to force any implemented debug acknowledge output signals to be asserted. The possible values of this bit are:

0 Debug acknowledge signals under normal processor control

1 Debug acknowledge signals asserted, regardless of the processor state.

For details of the recommended external debug interface, see *Run-control and cross-triggering signals* on page AppxA-5 and *DBGACK and DBGCPUDONE* on page AppxA-7.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

UDCCdis、ビット [12]

通信チャネルへのユーザモードアクセス ディセーブルビット。このビットの可能な値は次のとおりです。

0 通信チャネルへのユーザモード アクセスが可能です。

1 通信チャネルへのユーザモード アクセスは不可能です。

このビットが 1 にセットされているとき、ユーザモードのプロセスが CP14 操作によって DBGDIDR、DBGDSCRint、DBGDTRRXint、DBGDTRTXint のいずれかにアクセスを試みると、未定義命令例外が取得されます。このビットを 1 にセットすると、ユーザモードからすべての CP14 デバッグレジスタへのアクセスが禁止されます。

INTdis、ビット [11]

割り込みディセーブルビット。このビットは、IRQ と FIQ の取得をマスクするために使用できます。このビットの可能な値は次のとおりです。

0 割り込みが可能です。

1 割り込みは不可能です。

外部デバッガが、デバッグプロセスの一部として、非デバッグ状態でコードの一部を実行する必要があるが、そのコードの途中で割り込みが発生してはならない場合、外部デバッガはこのビットを 1 にセットできます。

例えば、定期的なタイマ割り込みが発生するシステムでコードのシングルステップ実行を行う場合、割り込みの周期はデバッガのステップ実行周期より短いことが考えられます。この場合、ステップ実行中に INTdis ビットを 1 にセットせず、デバッガがターゲットをステップ実行したと仮定すると、割り込みが保留されます。これは、CPSR で割り込みが可能であれば、プロセッサがデバッグ状態を終了した直後に割り込みが取得されることを意味します。

次のいずれかの条件が当てはまる場合、INTdis ビットは無視されます。

- DBGDSCR[15:14] == 0b00
- **DBGEN** が LOW である。

デバッグ認証信号の詳細については、第 C2 章*「侵襲性デバッグの認証」*を参照して下さい。

注

ISR が実装されている場合、ISR は INTdis ビットの値にかかわらず、IRQ および FIQ 信号のステータスを常に反映します。詳細については、*c12、割り込みステータスレジスタ (ISR)* P. B3- 150 を参照して下さい。

DBGack、ビット [10]

強制デバッグ応答ビット。デバッガは、このビットを使用して、実装されている任意のデバッグ応答出力信号を強制的にアサートできます。このビットの可能な値は次のとおりです。

0 デバッグ応答信号は、通常のプロセッサ制御下にあります。

1 デバッグ応答信号が、プロセッサの状態にかかわらずアサートされます。

推奨される外部デバッグインタフェースの詳細については、*実行制御およびクロストリガ信号* P. 付録 A- 5 および *DBGACK および DBGCPUDONE* P. 付録 A- 7 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If a debugger sets this bit to 1, it can then cause the processor to execute instructions in Non-debug state, while the rest of the system behaves as if the processor is in Debug state.

Note

The effect of setting DBGack to 1 takes no account of the **DBGEN** and **SPIDEN** signals. This means it asserts the debug acknowledge signals regardless of the invasive debug authentication settings.

Bit [9], v7 Debug

Reserved, UNK/SBZP.

DBGnoPWRDWN, bit [9], v6 Debug and v6.1 Debug only

Debug no power-down bit. This bit can be used to drive a debug no power-down output signal, **DBGNOPWRDWN**. The possible values of this bit are:

0 **DBGNOPWRDWN** driven LOW

1 **DBGNOPWRDWN** driven HIGH.

Note

- In v6 Debug and v6.1 Debug, this bit is not defined, but many implementations define DBGDSCR[9] as the DBGnoPWRDWN bit. If this bit is not implemented, DBGDSCR[9] is UNK/SBZP.
- In v7 Debug this use of this bit is replaced by the DBGnoPWRDWN bit in the DBGPRCR, see *Device Power-down and Reset Control Register (DBGPRCR), v7 Debug only* on page C10-31.

UND_I, bit [8], v6.1 Debug and v7 Debug

Sticky Undefined Instruction bit. This flag is set to 1 by any Undefined Instruction exceptions generated by instructions issued to the processor while in Debug state. The possible values of this bit are:

0 No Undefined Instruction exception has been generated since the last time this bit was cleared to 0

1 An Undefined Instruction exception has been generated since the last time this bit was cleared to 0.

The method of clearing this flag to 0, and the behavior of the flag, depends on the version of the Debug architecture:

v6.1 Debug

This flag is cleared to 0 when the external debugger reads the DBGDSCR.

v7 Debug This flag is cleared to 0 only by writing to bit [2] of the DBGDRCR, see *Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29.

Leaving Debug state with this flag set to 1 causes UNPREDICTABLE behavior.

When the processor is in Non-debug state, this flag is not set to 1 by an Undefined Instruction exception.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

デバッガは、このビットを 1 にセットした後で、プロセッサに対して命令を非デバッグ状態で実行させることができますが、システムの他の部分はプロセッサがデバッグ状態であるかのように動作します。

注

DBGack を 1 にセットした場合の影響には、DBGEN および SPIDEN 信号は関係しません。つまり、侵襲性デバッグ認証の設定にかかわらずデバッグ応答信号をアサートします。

ビット [9]、v7 デバッグのみ

予約、UNK/SBZP

DBGnoPWRDWN、ビット [9]、v6 デバッグおよび v6.1 デバッグのみ

デバッグ電力オフなしビット。このビットは、デバッグ電力オフなし出力信号 (DBGNOPWRDWN) を駆動するために使用できます。このビットの可能な値は次のとおりです。

0 DBGNOPWRDWN は LOW に駆動されています。

1 DBGNOPWRDWN は HIGH に駆動されています。

注

- v6 デバッグと v6.1 デバッグではこのビットは定義されていませんが、多くの実装で DBGDSCR[9] が DBGnoPWRDWN ビットとして定義されています。このビットが実装されていない場合、DBGDSCR[9] は UNK/SBZP です。
- v7 デバッグでは、このビットの使用は、DBGPRCR の DBGnoPWRDWN ビットに置き換えられています。*デバイス電力オフおよびリセット制御レジスタ (DBGPRCR)、v7 デバッグのみ* P. C10-31 を参照して下さい。

UND_I、ビット [8]、v6.1 デバッグおよび v7 デバッグのみ

スティッキー未定義命令ビット。このフラグは、デバッグ状態でプロセッサに発行された命令により生成された、未定義命令例外によって 1 にセットされます。このビットの可能な値は次のとおりです。

0 このビットが最後に 0 にクリアされた後で、未定義命令例外が生成されていません。

1 このビットが最後に 0 にクリアされた後で、未定義命令例外が生成されました。

このフラグを 0 にクリアする方法と、フラグの動作は、デバッグアーキテクチャのバージョンによって異なります。

v6.1 デバッグ

外部デバッガが DBGDSCR を読み出すと、このフラグが 0 にクリアされます。

v7デバッグ DBGDRCR のビット [2] に書き込みを行った場合のみ、このフラグが 0 にクリアされます。*デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10-29 を参照して下さい。

このフラグが 1 にセットされたままデバッグ状態を終了すると、予測不能な動作を引き起こします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For more information, see *Exceptions in Debug state* on page C5-20.

Bit [8], v6 Debug only

Reserved, UNK/SBZP.

ADABORT_l, bit [7], v7 Debug

Sticky Asynchronous Data Abort bit. This flag is set to 1 by any asynchronous abort that occurs when the processor is in Debug state and is discarded because the ADAdiscard bit, bit [19], is set to 1. The possible values of this bit are:

0 No asynchronous abort has been discarded since the last time this bit was cleared to 0

1 An asynchronous abort has been discarded since the last time this bit was cleared to 0.

This flag is cleared to 0 only by writing to bit [2] of the DBGDRCR, see *Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29.

Leaving Debug state with this flag set to 1 causes UNPREDICTABLE behavior.

When the processor is in Non-debug state this flag is never set to 1 when an asynchronous abort occurs.

For more information, see *Asynchronous aborts and entry to Debug state* on page C5-5 and *Exceptions in Debug state* on page C5-20.

ADABORT_l, bit [7], v6 Debug and v6.1 Debug

Sticky Asynchronous Data Abort bit. This flag is set to 1 by any asynchronous abort that occurs when the processor is in Debug state. The possible values of this bit are:

0 No asynchronous abort has occurred since the last time this bit was cleared to 0

1 An asynchronous abort has occurred since the last time this bit was cleared to 0.

This flag is cleared to 0 when the external debugger reads the DBGDSCR.

Some aspects of the behavior of this flag depend on the version of the Debug architecture:

v6.1 Debug

If the processor is in Non-debug state this flag is not set to 1 on an asynchronous abort.

v6 Debug The value of this flag is UNKNOWN when either the processor is in Non-debug state, or the ITRen bit, bit [13], is not set to 1.

For more information, see *Asynchronous aborts and entry to Debug state* on page C5-5 and *Exceptions in Debug state* on page C5-20.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

プロセッサが非デバッグ状態のとき、未定義命令例外が発生しても、このフラグは 1 にセットされません。

詳細については、*デバッグ状態での例外* P. C5- 20 を参照して下さい。

ビット [8]、v6 デバッグのみ

予約、UNK/SBZP

ADABORT_l、ビット [7]、v7 デバッグのみ

スティッキー非同期データアボート ビット。このフラグは、プロセッサがデバッグ状態のときに発生し、ADAdiscard（ビット [19]）が 1 にセットされていたため破棄された非同期アボートによって 1 にセットされます。このビットの可能な値は次のとおりです。

0 このビットが最後に 0 にクリアされた後で、非同期アボートが破棄されていません。

1 このビットが最後に 0 にクリアされた後で、非同期アボートが破棄されました。

DBGDRCR のビット [2] に書き込みを行った場合のみ、このフラグが 0 にクリアされます。*デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10- 29 を参照して下さい。

このフラグが 1 にセットされたままデバッグ状態を終了すると、予測不能な動作を引き起こします。

プロセッサが非デバッグ状態のとき、非同期アボートが発生しても、このフラグが 1 にセットされることはありません。

詳細については、*非同期アボートとデバッグ状態への遷移* P. C5- 5 と *デバッグ状態での例外* P. C5- 20 を参照して下さい。

ADABORT_l、ビット [7]、v6 デバッグおよび v6.1 デバッグのみ

スティッキー非同期データアボート ビット。このフラグは、プロセッサがデバッグ状態のときに非同期アボートが発生すると 1 にセットされます。このビットの可能な値は次のとおりです。

0 このビットが最後に 0 にクリアされた後で、非同期アボートが発生していません。

1 このビットが最後に 0 にクリアされた後で、非同期アボートが発生しました。

外部デバッガが DBGDSCR を読み出すと、このフラグが 0 にクリアされます。

このフラグの動作の一部は、デバッグアーキテクチャのバージョンに依存します。

v6.1 デバッグ

プロセッサが非デバッグ状態のとき、非同期アボートが発生しても、このフラグは 1 にセットされません。

v6デバッグ プロセッサが非デバッグ状態のとき、または ITRen ビット（ビット [13]）が 1 にセットされていないとき、このフラグの値は未知です。

詳細については、*非同期アボートとデバッグ状態への遷移* P. C5- 5 と *デバッグ状態での例外* P. C5- 20 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

SDABORT_l, bit [6]

Sticky Synchronous Data Abort bit. This flag is set to 1 by any Data Abort exception that is generated by a synchronous data abort when the processor is in Debug state. The possible values of this bit are:

0 No Data Abort exception has been generated by a synchronous data abort since the last time this bit was cleared to 0

1 A Data Abort exception has been generated by a synchronous data abort since the last time this bit was cleared to 0.

The behavior of the DBGITR depends on the value of the SDABORT_l bit, see *Instruction Transfer Register (DBGITR)* on page C10-46.

The method of clearing this flag to 0 depends on the version of the Debug architecture:

v7 Debug This flag is cleared to 0 only by writing to bit [2] of the DBGDRCR, see *Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29.

ARMv6 This flag is cleared to 0 when the external debugger reads the DBGDSCR.

Some aspects of the behavior of this flag depend on the version of the Debug architecture:

v7 Debug If the processor is in Non-debug state this flag is not set to 1 on a synchronous Data Abort exception.

Leaving Debug state with this flag set to 1 causes UNPREDICTABLE behavior.

v6.1 Debug

If the processor is in Non-debug state this flag is not set to 1 on a synchronous Data Abort exception.

v6 Debug If the processor is in Non-debug state, the value of this flag is UNKNOWN.

For more information, see *Exceptions in Debug state* on page C5-20.

MOE, bits [5:2]

Method of Debug Entry field. The permitted values of this field depend on the Debug architecture. For details of this field see *Method of Debug entry* on page C10-26.

RESTARTED, bit [1]

Processor Restarted bit. The possible values of this bit are:

0 The processor is exiting Debug state. This bit only reads as 0 between receiving a restart request, and restarting Non-debug state operation.

1 The processor has exited Debug state. This bit remains set to 1 if the processor re-enters Debug state.

After making a restart request, the debugger can poll this bit until it is set to 1. At that point it knows that the restart request has taken effect and the processor has exited Debug state.

Note

Polling the HALTED bit until it is set to 0 is not safe because the processor could re-enter Debug state as a result of another debug event before the debugger samples the DBGDSCR.

See Chapter C5 *Debug State* for a definition of Debug state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

SDABORT_l、ビット [6]

スティッキー同期データアボート ビット。このフラグは、プロセッサがデバッグ状態のときに、同期データアボートにより生成されたデータアボート例外により、1 にセットされます。このビットの可能な値は次のとおりです。

0 このビットが最後に 0 にクリアされた後で、同期データアボートによりデータアボート例外が生成されていません。

1 このビットが最後に 0 にクリアされた後で、同期データアボートによりデータアボート例外が生成されました。

DBGITR の動作は、SDABORT_l ビットの値に依存します。*命令転送レジスタ (DBGITR)* P. C10- 46 を参照して下さい。

このフラグを 0 にクリアする方法は、デバッグアーキテクチャのバージョンによって異なります。

v7デバッグ DBGDRCR のビット [2] に書き込みを行った場合のみ、このフラグが 0 にクリアされます。*デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10- 29 を参照して下さい。

ARMv6 外部デバッガが DBGDSCR を読み出すと、このフラグが 0 にクリアされます。

このフラグの動作の一部は、デバッグアーキテクチャのバージョンに依存します。

v7デバッグ プロセッサが非デバッグ状態のとき、同期データアボート例外が発生しても、このフラグは 1 にセットされません。

このフラグが 1 にセットされたままデバッグ状態を終了すると、予測不能な動作を引き起こします。

v6.1 デバッグ

プロセッサが非デバッグ状態のとき、同期データアボート例外が発生しても、このフラグは 1 にセットされません。

v6デバッグ プロセッサが非デバッグ状態のとき、このフラグの値は未知です。

詳細については、*デバッグ状態での例外* P. C5- 20 を参照して下さい。

MOE、ビット [5:2]

デバッグ開始方法フィールド。このフィールドに許可される値は、デバッグアーキテクチャに依存します。このフィールドの詳細については、*デバッグ開始の方法* P. C10- 26 を参照して下さい。

RESTARTED、ビット [1]

プロセッサ再始動ビット。このビットの可能な値は次のとおりです。

0 プロセッサがデバッグ状態を終了します。このビットは、再始動要求を受信してから、非デバッグ状態の動作を再開するまでの間のみ、0 として読み出されます。

1 プロセッサはデバッグ状態を終了しました。プロセッサが再度デバッグ状態に遷移しても、このビットは 1 にセットされたままです。

再始動要求を行った後で、デバッガはこのビットが 1 にセットされるまでポーリングを行うことができます。このビットが 1 にセットされた時点で、再始動要求が実行され、プロセッサがデバッグ状態を終了したことを確認できます。

注

HALTED ビットが 0 にセットされるまでポーリングを行うのは安全ではありません。これは、デバッガが DBGDSCR のサンプリングを行う前に、別のデバッグイベントの結果としてプロセッサが再度デバッグ状態に遷移する可能性があるためです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

HALTED, bit [0]

Processor Halted bit. The possible values of this bit are:

0 The processor is in Non-debug state.

1 The processor is in Debug state.

Note

Between receiving a restart request and restarting Non-debug state operation, the processor is in Debug state and this bit reads as 1

After programming a debug event, the external debugger can poll this bit until it is set to 1. At that point it knows that the processor has entered Debug state.

See Chapter C5 *Debug State* for a definition of Debug state.

Table C10-1 shows the access to each field of the DBGDSCR, and the reset value of each field. It also shows the Debug architecture versions in which each field is defined.

Table C10-1 DBGDSCR bit access and reset values

Bits	Field name	Version	Access[a]	Reset value[b]
[31]	-	-	UNK/SBZP	-
[30]	RXfull	All	Read-only	0
[29]	TXfull	All	Read-only	0
[28]	-	-	UNK/SBZP	-
[27]	RXfull_l	v7 Debug	Read-only	0
[26]	TXfull_l	v7 Debug	Read-only	0
[25]	PipeAdv	v7 Debug	Read-only	UNKNOWN
[24]	InstrCompl_l	v7 Debug	Read-only	UNKNOWN
[23:22]	-	-	UNK/SBZP	-
[21:20]	ExtDCCmode	v7 Debug	Read/write	00
[19]	ADAdiscard	v6.1 Debug, v7 Debug	Read-only or Read/write[c]	0
[18]	NS	v6.1 Debug, v7 Debug	Read-only	[f]
[17]	SPNIDdis	v6.1 Debug, v7 Debug	Read-only	[f]
[16]	SPIDdis	v6.1 Debug, v7 Debug	Read-only	[f]
[15]	MDBGen	All	RW_{Int}	0

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

デバッグ状態の定義については、第 C5 章*「デバッグ状態」*を参照して下さい。

HALTED、ビット [0]

プロセッサ停止ビット。このビットの可能な値は次のとおりです。

0 プロセッサは非デバッグ状態です。

1 プロセッサはデバッグ状態です。

注

再始動要求を受信してから非デバッグ状態の動作を再開するまでの間、プロセッサはデバッグ状態で、このビットは 1 として読み出されます。

デバッグイベントをプログラムした後で、外部デバッガはこのビットが 1 にセットされるまでポーリングを行うことができます。このビットが 1 にセットされた時点で、プロセッサがデバッグ状態に遷移したことを確認できます。

デバッグ状態の定義については、第 C5 章*「デバッグ状態」*を参照して下さい。

DBGDSCR の各フィールドに対するアクセスと、各フィールドのリセット時の値を、表 C10-1 に示します。この表には、それぞれのフィールドが定義されているデバッグアーキテクチャのバージョンも示されています。

表 C10-1 DBGDSCR ビットのアクセスとリセット時の値

ビット	フィールド名	バージョン	アクセス[a]	リセット時の値[b]
[31]	–	–	UNK/SBZP	–
[30]	RXfull	すべて	読み出し専用	0
[29]	TXfull	すべて	読み出し専用	0
[28]	–	–	UNK/SBZP	–
[27]	RXfull_l	v7 デバッグ	読み出し専用	0
[26]	TXfull_l	v7 デバッグ	読み出し専用	0
[25]	PipeAdv	v7 デバッグ	読み出し専用	未知
[24]	InstrCompl_l	v7 デバッグ	読み出し専用	未知
[23:22]	–	–	UNK/SBZP	–
[21:20]	ExtDCCmode	v7 デバッグ	読み出し / 書き込み	00
[19]	ADAdiscard	v6.1デバッグ、v7デバッグ	読み出し専用または読み出し / 書き込み[c]	0
[18]	NS	v6.1デバッグ、v7デバッグ	読み出し専用	[f]
[17]	SPNIDdis	v6.1デバッグ、v7デバッグ	読み出し専用	[f]
[16]	SPIDdis	v6.1デバッグ、v7デバッグ	読み出し専用	[f]

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C10-1 DBGDSCR bit access and reset values (continued)

Bits	Field name	Version	Access[a]	Reset value[b]
[14]	HDBGen	All	RW_{Ext}	0
[13]	ITRen	All	RW_{Ext}	0
[12]	UDCCdis	All	RW_{Int}	0
[11]	INTdis	All	RW_{Ext}	0
[10]	DBGack	All	RW_{Ext}	0
[9]	DBGnoPWRDWN	v6 Debug, v6.1 Debug[d]	RW_{Ext}	0
[8]	UND_l	v6.1 Debug, v7 Debug	Read-only[e]	0
[7]	ADABORT_l	All	Read-only[e]	0
[6]	SDABORT_l	All	Read-only[e]	0
[5:2]	MOE	All	RW_{Int}	0
[1]	RESTARTED	All	Read-only	[f]
[0]	HALTED	All	Read-only	[f]

a. For more information, including the meaning of RW_{Int} and RW_{Ext}, see *Access to the DBGDSCR* on page C10-27.
b. Debug logic reset value, the value after a debug logic reset.
c. The ADAdiscard bit can be read/write. This is IMPLEMENTATION DEFINED, see *Asynchronous aborts and entry to Debug state* on page C5-5.
d. For more information, see the v6 Debug and v6.1 Debug description of this field.
e. For details of how these bits are cleared to 0 see the descriptions of the bits. The method depends on the Debug architecture version.
f. These are read-only status bits that reflect the current state of the processor.

Access controls on the external view of the DCC registers and DBGITR, v7 Debug only

In v7 Debug, the DBGDSCR.ExtDCCmode field determines the external DCC access mode. This access mode, operating with flags in the DBGDSCR, controls all accesses made to DBGDTRRXext and DBGDTRTXext, and also controls some accesses to the Instruction Transfer Register DBGITR.

The DBGDSCR includes a ready flag and a latched ready flag for each of the registers DBGDTRRXext and DBGDTRTXext, and a latched ready flag for the DBGITR:

- RXfull and RXfull_l are the ready and latched ready flags for the DBGDTRRXext register
- TXfull and TXfull_l are the ready and latched ready flags for the DBGDTRTXext register
- InstrCompl_l is the latched ready flag for the DBGITR.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C10-1 DBGDSCR ビットのアクセスとリセット時の値（続き）

ビット	フィールド名	バージョン	アクセス[a]	リセット時の値[b]
[15]	MDBGen	すべて	RW_{Int}	0
[14]	HDBGen	すべて	RW_{Ext}	0
[13]	ITRen	すべて	RW_{Ext}	0
[12]	UDCCdis	すべて	RW_{Int}	0
[11]	INTdis	すべて	RW_{Ext}	0
[10]	DBGack	すべて	RW_{Ext}	0
[9]	DBGnoPWRDWN	v6デバッグ、v6.1デバッグ[d]	RW_{Ext}	0
[8]	UND_l	v6.1デバッグ、v7デバッグ	読み出し専用[e]	0
[7]	ADABORT_l	すべて	読み出し専用[e]	0
[6]	SDABORT_l	すべて	読み出し専用[e]	0
[5:2]	MOE	すべて	RW_{Int}	0
[1]	RESTARTED	すべて	読み出し専用	[f]
[0]	HALTED	すべて	読み出し専用	[f]

a. RW_{Int} および RW_{Ext} の意味など詳細については、*DBGDSCR へのアクセス* P. C10-27 を参照して下さい。
b. デバッグロジックのリセット値で、デバッグロジック リセット後の値。
c. ADAdiscard ビットは読み出し / 書き込みの場合もあります。これは実装定義です。*非同期アボートとデバッグ状態への遷移* P. C5-5 を参照して下さい。
d. 詳細については、このフィールドに関する v6 デバッグと v6.1 デバッグの説明を参照して下さい。
e. これらのビットが 0 にクリアされる方法の詳細については、ビットの説明を参照して下さい。方法は、デバッグアーキテクチャのバージョンによって異なります。
f. これらは読み出し専用のステータスビットで、プロセッサの現在の状態を反映します。

DCC レジスタと DBGITR の外部ビューへのアクセス制御、v7 デバッグのみ

v7 デバッグでは、DBGDSCR.ExtDCCmode フィールドにより、外部 DCC アクセスモードが決定されます。このアクセスモードは、DBGDSCR のフラグにより動作し、DBGDTRRXext と DBGDTRTXext に対して行われるすべてのアクセスを制御し、同時に命令転送レジスタ (DBGITR) への一部のアクセスも制御します。

DBGDSCR には、DBGDTRRXext と DBGDTRTXext の各レジスタについて準備完了フラグとラッチされた準備完了フラグが含まれており、DBGITR についてラッチされた準備完了フラグが含まれています。

- RXfull と RXfull_l は、DBGDTRRXext レジスタの準備完了フラグ、およびラッチされた準備完了フラグです。
- TXfull と TXfull_l は、DBGDTRTXext レジスタの準備完了フラグ、およびラッチされた準備完了フラグです。
- InstrCompl_l は、DBGITR のラッチされた準備完了フラグです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The ready flag for the DBGITR, InstrCompl, is an internal flag that cannot be accessed through any register.

For details of the ready state of each flag, and details of when the latched flags are updated, see the descriptions of the DBGDSCR flag bits.

Different external DCC access modes require the debugger to execute different sequences of accesses to the DBGDTRRXext and DBGDTRTXext registers, and to the DBGITR. This can affect the total number of accesses required.

Table C10-2 shows the three external DCC access modes:

Table C10-2 Meaning of the external DCC access mode values

DBGDSCR.ExtDCCmode	External DCC access mode	Description
0b00	Non-blocking mode	*Non-blocking mode*
0b01	Stall mode	*Stall mode* on page C10-23
0b10	Fast mode	*Fast mode* on page C10-23

Note

- Non-blocking mode is the default setting because improper use of the other modes can result in the external debug interface becoming deadlocked.
- For information that applies to all access modes see *Restrictions on accesses to DBGITR, DBGDTRRXext and DBGDTRTXext* on page C10-25.

See *Instruction and data transfer registers* on page C10-40. The external DCC access mode field has no effect on accesses to DBGDTRRXint and DBGDTRTXint.

Non-blocking mode

When Non-blocking mode is selected, reads from DBGDTRTXext and writes to DBGDTRRXext and DBGITR are ignored when the appropriate latched *ready* flag is not in the *ready* state:

- if RXfull_l is set to 1, writes to DBGDTRRXext are ignored
- if InstrCompl_l is set to 0, writes to DBGITR are ignored
- if TXfull_l is set to 0, reads from DBGDTRTXext are ignored and return an UNKNOWN value.

Following a successful write to DBGDTRRXext, RXfull and RXfull_l are set to 1.

Following a successful read from DBGDTRTXext, TXfull and TXfull_l are cleared to 0.

Following a successful write to DBGITR, InstrCompl and InstrCompl_l are cleared to 0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

DBGITR の準備完了フラグである InstrCompl は内部フラグで、どのレジスタ経由でもアクセスできません。

各フラグの準備完了状態の詳細と、ラッチされたフラグがどの時点で更新されるかの詳細については、DBGDSCR フラグビットの説明を参照して下さい。

それぞれの外部 DCC アクセスモードでは、デバッガが DBGDTRRXext および DBGDTRTXext レジスタと DBGITR に対して、異なるアクセスシーケンスを実行する必要があります。これは、必要なアクセスの総数に影響する可能性があります。

3 つの外部 DCC アクセスモードを、表 C10-2 に示します。

表 C10-2 外部 DCC アクセスモードの値の意味

DBGDSCR.ExtDCCmode	外部 DCC アクセスモード	説明
0b00	非ブロッキングモード	*非ブロッキングモード*
0b01	ストールモード	*ストールモード* P. C10-23
0b10	高速モード	*高速モード* P. C10-23

注

- デフォルト設定は非ブロッキングモードです。これは、他のモードを不適切に使用した場合、外部デバッグインタフェースがデッドロックに陥る可能性があるためです。
- すべてのアクセスモードに適用される情報については、*DBGITR*、*DBGDTRRXext*、*DBGDTRTXext へのアクセスの制限* P. C10-25 を参照して下さい。

命令およびデータ転送レジスタ P. C10-40 を参照して下さい。外部 DCC アクセスモードフィールドは、DBGDTRRXint と DBGDTRTXint へのアクセスには影響しません。

非ブロッキングモード

非ブロッキングモードが選択されている場合、DBGDTRTXext からの読み出しと、DBGDTRRXext および DBGITR への書き込みは、それらに対応するラッチされた *準備完了*フラグが *準備完了*状態でなければ無視されます。

- RXfull_l が 1 にセットされている場合、DBGDTRRXext への書き込みは無視されます。
- InstrCompl_l が 0 にセットされている場合、DBGITR への書き込みは無視されます。
- TXfull_l が 0 にセットされている場合、DBGDTRTXext からの読み出しは無視され、未知の値が返されます。

DBGDTRRXext への書き込みが成功した後で、RXfull と RXfull_l は 1 にセットされます。

DBGDTRTXext からの読み出しが成功した後で、TXfull と TXfull_l は 0 にクリアされます。

DBGITR への書き込みが成功した後で、InstrCompl と InstrCompl_l は 0 にクリアされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Debuggers accessing these registers must first read DBGDSCRext. This has the side-effect of copying RXfull and TXfull to RXfull_l and TXfull_l, and setting InstrCompl_l. The debugger can then use the returned value to determine whether a subsequent access to these registers will be ignored.

Stall mode

When Stall mode is selected, accesses to DBGDTRRXext, DBGDTRTXext, and DBGITR are modified such that each access stalls under the following conditions:

- writes to DBGDTRRXext are not completed until RXfull is 0
- writes to DBGITR are not completed until InstrCompl is 1
- reads from DBGDTRTXext are not completed until TXfull is 1.

If an access is stalled in this way you cannot access any of the debug registers until the stalled DBGDTRRXext, DBGDTRTXext, or DBGITR access completes. For more information about stalled accesses see *Stalling of accesses to the DCC registers* on page C10-25.

Following a write to DBGDTRRXext or DBGITR, or a read from DBGDTRTXext, the flags InstrCompl, InstrCompl_l, RXfull, RXfull_l, TXfull, and TXfull_l are set as in *Non-blocking mode* on page C10-22.

Note

The rules used in Non-blocking mode for ignoring accesses based on the values of the latched flags InstrCompl_l, RXfull_l and TXfull_l do not apply in Stall mode.

Stall mode can be selected when the processor is in Non-debug state. However, because Stall mode blocks the interface to the debug registers until the processor issues the correct `MCR` or `MRC` instruction to unblock the access, ARM recommends that you do not use Stall mode in cases where the external debugger does not have complete control over the instructions executing on the processor.

Accesses to DBGDTRRXext and DBGDTRTXext through the Extended CP14 interface are UNPREDICTABLE when Stall mode is selected.

Fast mode

If Fast mode is selected and the DBGDSCR.ITRen bit is 0, or the processor is in Non-debug state, the results are UNPREDICTABLE.

When Fast mode is selected, a write to the DBGITR does not trigger an instruction for execution. Instead, the instruction is latched. The latched value is retained until either a new value is written to the DBGITR, or the access mode is changed.

For accesses through the external debug interface or the memory-mapped interface:

- when an instruction is latched, any read of DBGDTRTXext or write to DBGDTRRXext causes the processor to execute the latched instruction
- when no instruction is latched, any access to DBGDTRRXext or DBGDTRTXext is UNPREDICTABLE.

Any access to DBGDTRRXext or DBGDTRTXext through the Extended CP14 interface is UNPREDICTABLE.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

これらのレジスタにアクセスするデバッガは、最初に DBGDSCRext を読み出す必要があります。この操作には、RXfull と TXfull を RXfull_l と TXfull_l にコピーし、InstrCompl_l をセットする副作用があります。その後でデバッガは、返された値を使用して、それらのレジスタに対する以後のアクセスが無視されるかどうかを判定できます。

ストールモード

ストールモードが選択されている場合、DBGDTRRXext、DBGDTRTXext、DBGITR へのアクセスは、次の条件で各アクセスがストールするように変更されます。

- RXfull が 0 になるまで、DBGDTRRXext への書き込みは完了しません。
- InstrCompl が 1 になるまで、DBGITR への書き込みは完了しません。
- TXfull が 1 になるまで、DBGDTRTXext からの読み出しは完了しません。

この方法でアクセスがストールした場合、ストールした DBGDTRRXext、DBGDTRTXext、または DBGITR へのアクセスが完了するまで、デバッグレジスタのいずれにもアクセスできません。ストールしたアクセスの詳細については、*DCC レジスタへのアクセスのストール* P. C10-25 を参照して下さい。

DBGDTRRXext または DBGITR への書き込み、または DBGDTRTXext からの読み出しの後で、InstrCompl、InstrCompl_l、RXfull、RXfull_l、TXfull、TXfull_l フラグは、*非ブロッキングモード* P. C10-22 に示されているようにセットされます。

注

非ブロッキングモードで、ラッチされたフラグ InstrCompl_l、RXfull_l、TXfull_l の値に基づいてアクセスを無視するために使用される規則は、ストールモードには適用されません。

ストールモードは、プロセッサが非デバッグ状態のときに選択できます。ただし、ストールモードでは、プロセッサが正しい `MCR` または `MRC` 命令を発行してアクセスのブロックを解除するまで、デバッグレジスタへのインタフェースがブロックされるため、プロセッサで実行される命令を外部デバッガが完全に制御していない場合にはストールモードを使用しないことをお勧めします。

ストールモードが選択されている場合、拡張 CP14 インタフェース経由の DBGDTRRXext および DBGDTRTXext へのアクセスは予測不能です。

高速モード

高速モードが選択されており、DBGDSCR.ITRen ビットが 0 の場合、またはプロセッサが非デバッグ状態の場合、結果は予測不能です。

高速モードが選択されている場合、DBGITR への書き込みは命令の実行をトリガしません。代わりに、その命令はラッチされます。ラッチされた値は、新しい値が DBGITR に書き込まれるか、アクセスモードが変更されるまで保持されます。

外部デバッグインタフェースまたはメモリマップされたインタフェース経由のアクセスについては、次の規則が適用されます。

- 命令がラッチされているとき、DBGDTRTXext の読み出しまたは DBGDTRRXext への書き込みが行われると、プロセッサは常にラッチされた命令を実行します。
- 命令がラッチされていないとき、DBGDTRRXext または DBGDTRTXext へのアクセスは予測不能です。

拡張 CP14 インタフェース経由での DBGDTRRXext または DBGDTRTXext へのアクセスは、すべて予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Fast mode enables a single instruction to be executed repeatedly, without reloading the DBGITR.

In Fast mode:

- Writes to DBGITR do not trigger an instruction to be executed. If a previously issued instruction is executing, it must not be affected by the write to the DBGITR. Implementations can choose to stall the write until InstrCompl is set to 1 to achieve this requirement.
- Writes to DBGDTRRXext:
  - — are not completed until InstrCompl is set to 1
  - — write the data to the DBGDTRRX Register
  - — issue the instruction last written to DBGITR. If the issued instruction reads from DBGDTRRXint, the instruction reads the value written to DBGDTRRXext by this write.

  If RXfull is set to 1 before the write, then after the write the values of DBGDTRRX and the RXfull and RXfull_l flags in the DBGDSCR are UNKNOWN.
- Reads from DBGDTRTXext:
  - — Are not completed until InstrCompl is set to 1.
  - — Return the data from the DBGDTRTX.
  - — Issue the instruction last written to the DBGITR. If the issued instruction writes to DBGDTRTXint, the instruction does not affect the value returned from this read of DBGDTRTXext. That is, this instruction can write the next DBGDTRTXext value to be read.

  If TXfull is set to 0 before the read, then after the read the values of DBGDTRTX and the TXfull and TXfull_l flags in the DBGDSCR are UNKNOWN.

If a Fast mode access is stalled you cannot access any of the debug registers until the stalled DBGDTRRXext, DBGDTRTXext, or DBGITR access completes. For more information about stalled accesses see *Stalling of accesses to the DCC registers* on page C10-25.

Note

The rules used in Non-blocking mode for ignoring accesses based on the values of the latched flags InstrCompl_l, RXfull_l and TXfull_l do not apply in Fast mode.

If the DBGDSCR.SDABORT_l bit is set to 1, reads of DBGDTRTXext and writes to DBGDTRTXext do not cause the latched instruction to be executed by the processor, and the access completes immediately. In these cases:

- reading DBGDTRTXext returns an UNKNOWN value, and the values of DBGDTRTX and the TXfull and TXfull_l flags become UNKNOWN
- if you write to DBGDTRRXext, the values of DBGDTRRX and the RXfull and RXfull_l flags in the DBGDSCR become UNKNOWN.

Otherwise, following a write to DBGDTRRXext or DBGITR, or a read from DBGDTRTXext, the flags InstrCompl, InstrCompl_l, RXfull, RXfull_l, TXfull, and TXfull_l are set as in *Non-blocking mode* on page C10-22.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

高速モードでは、DBGITR をリロードせず、単一の命令を繰り返し実行できます。

高速モードでは、次の規則が適用されます。

- DBGITR への書き込みにより、命令の実行がトリガされません。以前に発行された命令が実行中の場合、DBGITR への書き込みによりその命令が影響を受けないことが必要です。実装では、この条件を満たすため、InstrCompl が 1 にセットされるまで書き込みをストールすることを選択できます。
- DBGDTRRXext への書き込みは、次のように動作します。
  - – InstrCompl が 1 にセットされるまで完了しません。
  - – データを DBGDTRRX レジスタへ書き込みます。
  - – DBGITR に最後に書き込まれた命令を発行します。発行された命令が DBGDTRRXint から読み出しを行う場合、その命令によって読み出されるのは、この書き込みによって DBGDTRRXext に書き込まれた値です。

  書き込みの前に RXfull が 1 にセットされた場合、書き込みの後の DBGDTRRX、および DBGDSCR の RXfull および RXfull_l フラグの値は未知です。
- DBGDTRTXext からの読み出しは、次のように動作します。
  - – InstrCompl が 1 にセットされるまで完了しません。
  - – DBGDTRTX からのデータを返します。
  - – DBGITR に最後に書き込まれた命令を発行します。発行された命令が DBGDTRTXint に書き込みを行う場合、その命令は、この DBGDTRTXext の読み出しによって返される値に影響を及ぼしません。つまり、この命令が書き込む値は、次に DBGDTRTXext が読み出されたときに返されます。

  読み出しの前に TXfull が 0 にセットされた場合、読み出しの後の DBGDTRTX、および DBGDSCR の TXfull および TXfull_l フラグの値は未知です。

高速モードアクセスがストールした場合、ストールした DBGDTRRXext、DBGDTRTXext、または DBGITR へのアクセスが完了するまで、デバッグレジスタのいずれにもアクセスできません。ストールしたアクセスの詳細については、*DCC レジスタへのアクセスのストール* P. C10-25 を参照して下さい。

注

非ブロッキングモードで、ラッチされたフラグ InstrCompl_l、RXfull_l、TXfull_l の値に基づいてアクセスを無視するために使用される規則は、高速モードには適用されません。

DBGDSCR.SDABORT_l ビットが 1 にセットされている場合、DBGDTRTXext の読み出しと DBGDTRTXext への書き込みによって、ラッチされた命令がプロセッサにより実行されることはなく、アクセスは直ちに完了します。これらの場合、次の規則が適用されます。

- DBGDTRTXext の読み出しで返される値は未知で、DBGDTRTX と、TXfull および TXfull_l フラグは未知の値になります。
- DBGDTRRXext へ書き込みを行うと、DBGDTRRX と、DBGDSCR の RXfull および RXfull_l フラグは未知の値になります。

それ以外の場合、DBGDTRRXext または DBGITR への書き込み、または DBGDTRTXext からの読み出しの後で、InstrCompl、InstrCompl_l、RXfull、RXfull_l、TXfull、TXfull_l フラグは、*非ブロッキングモード* P. C10-22 に示されているようにセットされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Stalling of accesses to the DCC registers

In Stall mode and Fast mode, accesses to the DCC registers can stall:

- The mechanism by which an access is stalled by the external debug interface must be defined by the external debug interface. For details of how accesses are stalled by the recommended ARM Debug Interface v5, see the *ARM Debug Interface v5 Architecture Specification*.
- The mechanism by which an access is stalled by the memory-mapped interface must be defined by the memory-mapped interface.
- A stall is a side-effect of an access. If the debug logic is in a state where an access has no side-effects, the access does not stall. For more information about debug logic states in which accesses have no side effects see *Permission summaries for memory-mapped and external debug interfaces* on page C6-45.

Note

When the selected DCC access mode is Stall mode or Fast mode, all accesses through the Extended CP14 interface are UNPREDICTABLE.

Restrictions on accesses to DBGITR, DBGDTRRXext and DBGDTRTXext

If an access is made when the OS Lock is set or when the Sticky Power-down status bit is set to 1, then:

- the access generates an error response
- register reads have no side-effects
- register writes are ignored
- the flags remain unchanged.

This applies both to accesses through the external debug interface and to accesses through the memory-mapped interface.

If an access is made through the memory-mapped interface when the Software Lock is set then:

- register reads have no side-effects
- register writes are ignored
- the flags remain unchanged.

For more information, see *Permission summaries for memory-mapped and external debug interfaces* on page C6-45.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DCC レジスタへのアクセスのストール

ストールモードと高速モードでは、DCC レジスタへのアクセスは次の条件でストールすることがあります。

- 外部デバッグインタフェースによってアクセスがストールする機構は、外部デバッグインタフェースにより定義される必要があります。推奨される ARM デバッグインタフェース v5 でアクセスがストールする方法の詳細については、『*ARM デバッグインタフェース v5、アーキテクチャ仕様書*』を参照して下さい。
- メモリマップされたインタフェースによってアクセスがストールする機構は、メモリマップされたインタフェースにより定義される必要があります。
- ストールは、アクセスの副作用です。デバッグロジックが、アクセスの副作用が発生しない状態であれば、アクセスはストールしません。アクセスの副作用が発生しないデバッグロジックの状態の詳細については、*メモリマップおよび外部デバッグインタフェースのアクセス許可の概要* P. C6- 45 を参照して下さい。

注

選択されている DCC アクセスモードがストールモードまたは高速モードの場合、拡張 CP14 インタフェース経由のアクセスは、すべて予測不能です。

DBGITR、DBGDTRRXext、DBGDTRTXext へのアクセスの制限

OS ロックがセットされている、またはスティッキー電力オフステータス ビットが 1 にセットされているときにアクセスが行われると、次の動作が行われます。

- アクセスによりエラー応答が生成されます。
- レジスタ読み出しで副作用は発生しません。
- レジスタへの書き込みは無視されます。
- フラグは変更されません。

これは、外部デバッグインタフェース経由のアクセスと、メモリマップされたインタフェース経由のアクセスの両方に適用されます。

ソフトウェアロックがセットされているときに、メモリマップされたインタフェース経由でアクセスが行われると、次の動作が行われます。

- レジスタ読み出しで副作用は発生しません。
- レジスタへの書き込みは無視されます。
- フラグは変更されません。

詳細については、*メモリマップおよび外部デバッグインタフェースのアクセス許可の概要* P. C6- 45 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Method of Debug entry

The Method of Debug Entry is indicated by the DBGDSCR.MOE field. Table C10-3 shows the meanings of the possible values of the DBGDSCR.MOE field, and also shows:

- the versions of the Debug architecture for which each value is permitted
- the section where the corresponding method of entry is described.

Table C10-3 Meaning of Method of Debug Entry values

MOE bits	Debug versions	Debug entry caused by:	Section, notes
0000	All	Halt Request debug event	*Halting debug events* on page C3-38.
0001	All	Breakpoint debug event.	*Breakpoint debug events* on page C3-5.
0010	All	Asynchronous Watchpoint debug event.	*Watchpoint debug events* on page C3-15.
0011	All	BKPT Instruction debug event.	*BKPT Instruction debug events* on page C3-20.
0100	All	External Debug Request debug event.	*Halting debug events* on page C3-38.
0101	All	Vector Catch debug event.	*Vector Catch debug events* on page C3-20.
0110	v6 only	D-side abort.	This MOE value is reserved in v6.1 and v7.
0111	v6 only	I-side abort.	This MOE value is reserved in v6.1 and v7.
1000	v7	OS Unlock Catch debug event.	*Halting debug events* on page C3-38. This MOE value is reserved in v6 and v6.1.
1001	All	Reserved.	-
1010	v7	Synchronous Watchpoint debug event.	*Watchpoint debug events* on page C3-15 This MOE value is reserved in v6 and v6.1.
1011-1111	All	Reserved.	-

A Prefetch Abort or Data Abort handler can determine whether a debug event occurred by checking the value of the relevant Fault Status Register, IFSR or DFSR. It then uses the DBGDSCR.MOE bits to determine the specific debug event.

In v6 Debug, the DBGDSCR can be checked first to determine whether an abort has occurred, and hence whether the abort handler jumps to the debug monitor or not. In v6.1 Debug and v7 Debug the *D-side abort occurred* and *I-side abort occurred* encodings are reserved. Therefore, an abort handler must always check the IFSR or DFSR first.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

デバッグ開始の方法

デバッグ開始の方法は、DBGDSCR.MOE フィールドにより示されます。DBGDSCR.MOE フィールドの可能な値とその意味を、表 C10-3 に示します。この表には、次の内容も示されています。

- それぞれの値が許可されるデバッグアーキテクチャのバージョン
- 対応する開始方法が説明されているセクション

表 C10-3 デバッグ開始方法の値の意味

MOE ビット	デバッグ バージョン	デバッグ開始を引き起こした イベント	セクション、注
0000	すべて	ホールト要求デバッグイベント	*ホールトデバッグ イベント* P. C3-38
0001	すべて	ブレークポイントデバッグ イベント	*ブレークポイントデバッグ イベント* P. C3-5
0010	すべて	非同期ウォッチポイントデバッグ イベント	*ウォッチポイントデバッグ イベント* P. C3-15
0011	すべて	BKPT 命令デバッグイベント	*BKPT 命令デバッグイベント* P. C3-20
0100	すべて	外部デバッグ要求デバッグ イベント	*ホールトデバッグ イベント* P. C3-38
0101	すべて	ベクタキャッチ デバッグ イベント	*ベクタキャッチデバッグ イベント* P. C3-20
0110	v6 のみ	D 側アボート	MOE のこの値は、v6.1 と v7 では予約されています。
0111	v6 のみ	I 側アボート	MOE のこの値は、v6.1 と v7 では予約されています。
1000	v7	OS ロック解除キャッチデバッグ イベント	*ホールトデバッグ イベント* P. C3-38 MOE のこの値は、v6 と v6.1 では予約されています。
1001	すべて	予約	–
1010	v7	同期ウォッチポイントデバッグ イベント	*ウォッチポイントデバッグ イベント* P. C3-15 MOE のこの値は、v6 と v6.1 では予約されています。
1011–1111	すべて	予約	–

プリフェッチアボートまたはデータアボート ハンドラは、関連するフォールトステータス レジスタである IFSR または DFSR の値をチェックすることで、デバッグイベントが発生したかどうかを判定できます。その後で、DBGDSCR.MOE ビットを使用して、デバッグイベントを特定できます。

v6 デバッグでは、最初に DBGDSCR をチェックしてアボートが発生したかどうかを判定し、結果としてアボートハンドラがデバッグモニタにジャンプするかどうかを判定できます。v6.1 デバッグと v7 デバッグでは、*D 側アボート発生*および *I 側アボート発生*のエンコードは予約されています。したがって、アボートハンドラは常に IFSR または DFSR を最初にチェックする必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996–1998, 2000, 2004–2008 ARM Limited. All rights reserved.

When debug is disabled, and when debug events are not permitted, the BKPT instruction generates a debug exception rather than being ignored. The DBGDSCR, IFSR, and IFAR are set as if a BKPT Instruction debug exception occurred. See *Effects of debug exceptions on CP15 registers and the DBGWFAR* on page C4-4. For security reasons, monitor software might need to check that debug was enabled and that the debug event was permitted before communicating with an external debugger.

In v7 Debug support for synchronous watchpoint events is added, see *Synchronous and Asynchronous Watchpoint debug events* on page C3-18.

Access to the DBGDSCR

In the Access column of Table C10-1 on page C10-20, read/write bits in the DBGDSCR are indicated by either RW_{Int} or RW_{Ext}.

In v6 Debug and v6.1 Debug:

- the meanings of the RW_{Int} and RW_{Ext} indications are:
  - **RW_{Int}** The bit is read/write in the internal view of the register, and read-only in the external view.
  - **RW_{Ext}** The bit is read/write in the external view of the register, and read-only in the internal view.
- the internal view, DBGDSCRint, is accessed using coprocessor instructions
- the external view, DBGDSCRext, is accessed through the external debug interface.

In v7 Debug:

- all read/write bits, whether indicated by RW_{Int} or RW_{Ext}, are read/write in DBGDSCRext
- DBGDSCRext can be accessed through the Extended CP14 interface, the memory-mapped interface, and the external debug interface
- DBGDSCRint is read-only, and is accessed using coprocessor instructions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

デバッグが不可能な場合、およびデバッグイベントが禁止されている場合、BKPT 命令は無視されず、デバッグ例外を生成します。DBGDSCR、IFSR、IFAR は、BKPT 命令によるデバッグ例外が発生した場合と同様に設定されます。*CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響* P. C4-4 を参照して下さい。セキュリティ上の理由から、モニタソフトウェアは、外部デバッガとの通信を行う前に、デバッグが可能で、デバッグイベントが許可されていることをチェックすることが必要な可能性があります。

v7 デバッグでは、同期ウォッチポイントイベントのサポートが追加されています。*同期と非同期のウォッチポイントデバッグ イベント* P. C3-18 を参照して下さい。

DBGDSCR へのアクセス

P. C10-20 の表 C10-1 のアクセスの列では、DBGDSCR の読み出し / 書き込みビットは RW_{Int} または RW_{Ext} と記載されています。

v6 デバッグおよび v6.1 デバッグでは、次の規則が適用されます。

- RW_{Int} および RW_{Ext} という記述は、次の意味を持ちます。
  - RW_{Int} このビットはレジスタの内部ビューでは読み出し / 書き込み、外部ビューでは読み出し専用です。
  - RW_{Ext} このビットはレジスタの外部ビューでは読み出し / 書き込み、内部ビューでは読み出し専用です。
- 内部ビュー DBGDSCRint には、コプロセッサ命令を使用してアクセスします。
- 外部ビュー DBGDSCRext は、外部デバッグインタフェース経由でアクセスされます。

v7 デバッグでは、次の規則が適用されます。

- すべての読み出し / 書き込みビットは、RW_{Int} または RW_{Ext} のいずれの記述でも、DBGDSCRext での読み出し / 書き込みです。
- DBGDSCRext には、拡張 CP14 インタフェース、メモリマップされたインタフェース、および外部デバッグインタフェース経由でアクセスできます。
- DBGDSCRint は読み出し専用で、コプロセッサ命令を使用してアクセスされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.3.2 Watchpoint Fault Address Register (DBGWFAR)

The Watchpoint Fault Address Register, DBGWFAR, returns information about the address of the instruction that accessed a watchpointed address.

The DBGWFAR:

- is debug register 6 at offset `0x018`
- is a read/write register
- is implemented differently in different versions of the Debug architecture:

  v6 Debug In v6 Debug, the DBGWFAR can be accessed only through CP15.

  v6.1 Debug
  In v6.1 Debug, the DBGWFAR can be accessed through the debug register interfaces, and using the CP15 access is deprecated.

  v7 Debug In v7 Debug, the CP15 encoding used for the DBGWFAR in v6 Debug is UNDEFINED in User mode and UNPREDICTABLE in privileged modes. The DBGWFAR can be accessed only through the debug register interfaces.

The format of the DBGWFAR is:

31 … 0
(Instruction address) + offset

(Instruction address) + offset, bits [31:0]

When Watchpoint debug events are permitted, on every Watchpoint debug event the DBGWFAR is updated with the address of the instruction that accessed the watchpointed address plus an offset that depends on the processor instruction set state when the instruction was executed:

- 8 if the processor was in ARM state
- 4 if the processor was in Thumb or ThumbEE state
- an IMPLEMENTATION DEFINED offset if the processor was in Jazelle state.

See *Memory addresses* on page C3-23 for a definition of the *Instruction Virtual Address* (IVA) used to update the DBGWFAR.

The debug logic reset value of the DBGWFAR is UNKNOWN.

A processor with a trivial implementation of the Jazelle extension can implement DBGWFAR[0] as RAZ/WI, see *Trivial implementation of the Jazelle extension* on page B1-81 for more information. In such an implementation, software must use a SBZP policy when writing to DBGWFAR[0].

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.3.2 ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)

ウォッチポイントフォールトアドレス レジスタ (DBGWFAR) は、ウォッチポイントアドレスにアクセスした命令のアドレスについての情報を返します。

DBGWFAR には次の特徴があります。

- デバッグレジスタ 6、オフセット `0x018`
- 読み出し / 書き込みレジスタ
- デバッグアーキテクチャのバージョンによって実装が異なります。

  v6デバッグ v6 デバッグでは、CP15 経由でのみ DBGWFAR にアクセスできます。

  v6.1 デバッグ
  v6.1 デバッグでは、デバッグレジスタ インタフェース経由で DBGWFAR にアクセスでき、CP15 アクセスの使用は推奨されません。

  v7デバッグ v7 デバッグでは、v6 デバッグで DBGWFAR に使用されている CP15 エンコードは、ユーザモードでは未定義、特権モードでは予測不能です。デバッグレジスタ インタフェース経由でのみ DBGWFAR にアクセスできます。

DBGWFAR のフォーマットは次のとおりです。

31 … 0
(命令のアドレス) + オフセット

(命令のアドレス) + オフセット、ビット [31:0]

ウォッチポイントデバッグ イベントが許可されているとき、すべてのウォッチポイントデバッグ イベントについて、DBGWFAR は、ウォッチポイントアドレスにアクセスした命令のアドレスに、その命令が実行されたときのプロセッサの命令セット状態によって異なるオフセットを加えた値に更新されます。

- プロセッサが ARM 状態の場合は 8
- プロセッサが Thumb または ThumbEE 状態の場合は 4
- プロセッサが Jazelle 状態の場合は実装定義のオフセット

DBGWFAR の更新に使用される *命令仮想アドレス* (IVA) の定義については、*メモリアドレス* P. C3- 23 を参照して下さい。

デバッグロジック リセット時の DBGWFAR の値は未知です。

Jazelle 拡張機能のトリビアル実装を搭載しているプロセッサでは、DBGWFAR[0] を RAZ/WI として実装できます。詳細については、*Jazelle 拡張機能のトリビアル実装* P. B1- 81 を参照して下さい。このような実装では、ソフトウェアは DBGWFAR[0] に書き込むときに SBZP ポリシーを使用する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.3.3 Debug Run Control Register (DBGDRCR), v7 Debug only

The Debug Run Control Register, DBGDRCR, requests the processor to enter or leave Debug state. It is also used to clear to 0 the sticky exception bits in the DBGDSCR.

The DBGDRCR is:

- debug register 36, at offset `0x090`
- a write-only register
- implemented only in v7 Debug
- when the Security Extensions are implemented, a Common register.

In v6 Debug and v6.1 Debug, register 36 is not defined.

The format of the DBGDRCR is:

Bits [31:5] Reserved, SBZ.

Cancel BIU Requests, bit [4]

Cancel Bus Interface Unit Requests bit. The actions on writing to this bit are:

0 no action

1 cancel pending accesses.

See *Cancel Bus Interface Unit (BIU) Requests* on page C10-30. It is IMPLEMENTATION DEFINED whether this feature is supported. If this feature is not implemented, writes to this bit are ignored.

It is UNPREDICTABLE whether a write of 1 to this bit has any effect when the processor is powered-down.

Clear Sticky Pipeline Advance flag, bit [3]

This bit is used to clear the DBGDSCR.PipeAdv bit, the Sticky Pipeline Advance bit, to 0. The actions on writing to this bit are:

0 no action

1 clear the DBGDSCR.PipeAdv bit to 0.

When the processor is powered down, it is UNPREDICTABLE whether a write of 1 to this bit clears DBGDSCR.PipeAdv to 0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.3.3 デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ

デバッグ実行制御レジスタ (DBGDRCR) は、デバッグ状態の開始または終了をプロセッサに要求します。DBGDSCR のスティッキー例外ビットを 0 にクリアするためにも使用されます。

DBGDRCR の特徴は次のとおりです。

- デバッグレジスタ 36、オフセット `0x090`
- 書き込み専用レジスタ
- v7 デバッグにのみ実装されています。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

v6 デバッグおよび v6.1 デバッグでは、レジスタ 36 は定義されていません。

DBGDRCR のフォーマットは次のとおりです。

ビット [31:5] 予約、SBZ

BIU 要求のキャンセル、ビット [4]

バスインタフェース ユニット要求キャンセルビット。このビットへの書き込みは、次の動作を行います。

0 動作を行いません。

1 保留中のアクセスをキャンセルします。

バスインタフェース ユニット (BIU) 要求のキャンセル P. C10-30 を参照して下さい。この機能をサポートするかどうかは実装定義です。この機能が実装されていない場合、このビットへの書き込みは無視されます。

プロセッサの電力がオフになっているとき、このビットに 1 を書き込むことで影響があるかどうかは予測不能です。

スティッキーパイプライン アドバンスフラグのクリア、ビット [3]

このビットは、DBGDSCR.PipeAdv ビット（スティッキーパイプラインアドバンス ビット）を 0 にクリアするために使用されます。このビットへの書き込みは、次の動作を行います。

0 動作を行いません。

1 DBGDSCR.PipeAdv ビットを 0 にクリアします。

プロセッサの電力がオフになっているとき、このビットに 1 を書き込むことで DBGDSCR.PipeAdv が 0 にクリアされるかどうかは予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Clear Sticky Exceptions flags, bit [2]

This bit is used to clear the sticky exceptions flags in the DBGDSCR to 0. The actions on writing to this bit are:

0 no action

1 clear DBGDSCR[8:6] to 0b000.

Writing 1 to this bit clears the DBGDSCR.UND_l, DBGDSCR.ADABORT_l, and DBGDSCR.SDABORT_l sticky exceptions flags, DBGDSCR[8:6], to 0b000.

When the processor is in Non-debug state, it is UNPREDICTABLE whether a write of 1 to this bit clears DBGDSCR[8:6] to 0b000.

When the processor is in Debug state, it can leave Debug state by performing a a single write to DBGDRCR with DBGDRCR[2:1] = 0b11. This:

- clears DBGDSCR[8:6] to 0b000
- requests exit from Debug state.

Restart request, bit [1]

Restart request bit. The actions on writing to this bit are:

0 no action

1 request exit from Debug state.

Writing 1 to this bit requests that the processor leaves Debug state. This request is held until the processor exits Debug state.

Once the request has been made, the debugger can poll the DBGDSCR.RESTARTED bit until it reads 1.

Writes to this bit are ignored if the processor is in Non-debug state.

Halt request, bit [0]

Halt request bit. The actions on writing to the this bit are:

0 no action

1 request entry to Debug state.

Writing 1 to this bit requests that the processor enters Debug state. This request is held until the processor enters Debug state, see *Halting debug events* on page C3-38.

Once the request has been made, the debugger can poll the DBGDSCR.HALTED bit until it reads 1.

Writes to this bit are ignored if the processor is already in Debug state.

Cancel Bus Interface Unit (BIU) Requests

When support for Cancel BIU Requests is implemented, if 1 is written to the Cancel BIU Requests bit, the processor cancels any pending Bus Interface Unit Request accesses until Debug state is entered. This means it cancels any pending accesses to the system bus. When this request is made an implementation must abandon all data load and store accesses. It is IMPLEMENTATION DEFINED whether other accesses, including instruction fetches and cache operations, are also abandoned.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

スティッキー例外フラグのクリア、ビット [2]

このビットは、DBGDSCR のスティッキー例外フラグを 0 にクリアするために使用されます。このビットへの書き込みは、次の動作を行います。

0 動作を行いません。

1 DBGDSCR[8:6] を 0b000 にクリアします。

このビットに 1 を書き込むと、DBGDSCR.UND_l、DBGDSCR.ADABORT_l、および DBGDSCR.SDABORT_l スティッキー例外フラグ（DBGDSCR[8:6]）が 0b000 にクリアされます。

プロセッサが非デバッグ状態のとき、このビットに 1 を書き込むことで DBGDSCR[8:6] が 0b000 にクリアされるかどうかは予測不能です。

プロセッサがデバッグ状態のとき、DBGDRCR に対して DBGDRCR[2:1] = 0b11 の単一の書き込みを実行すると、プロセッサをデバッグ状態のままに維持できます。これにより、次の動作が実行されます。

- DBGDSCR[8:6] を 0b000 にクリアします。
- デバッグ状態の終了を要求します。

再始動要求、ビット [1]

再始動要求ビット。このビットへの書き込みは、次の動作を行います。

0 動作を行いません。

1 デバッグ状態の終了を要求します。

このビットに 1 を書き込むと、プロセッサに対してデバッグ状態の終了を要求します。この要求は、プロセッサがデバッグ状態を終了するまで保持されます。

要求が行われた後で、デバッガは DBGDSCR.RESTARTED ビットの読み出し値が 1 になるまでポーリングを行うことができます。

プロセッサが非デバッグ状態の場合、このビットへの書き込みは無視されます。

ホールト要求、ビット [0]

ホールト要求ビット。このビットへの書き込みは、次の動作を行います。

0 動作を行いません。

1 デバッグ状態の開始を要求します。

このビットに 1 を書き込むと、プロセッサに対してデバッグ状態の開始を要求します。この要求は、プロセッサがデバッグ状態に遷移するまで保持されます。*ホールトデバッグ イベント* P. C3- 38 を参照して下さい。

要求が行われた後で、デバッガは DBGDSCR.HALTED ビットの読み出し値が 1 になるまでポーリングを行うことができます。

プロセッサがすでにデバッグ状態の場合、このビットへの書き込みは無視されます。

バスインタフェース ユニット (BIU) 要求のキャンセル

BIU 要求キャンセルのサポートが実装されている場合、BIU 要求キャンセルビットに 1 が書き込まれると、プロセッサはデバッグ状態に遷移するまで、保留中のバスインタフェース ユニット要求アクセスをすべてキャンセルします。これは、システムバスへの保留中のアクセスをすべてキャンセルすることを意味します。この要求が発行されたとき、実装はデータのロードおよびストアアクセスをすべて破棄する必要があります。命令フェッチやキャッシュ操作を含む他のアクセスも破棄されるかどうかは、実装定義です。

デバッグ状態の開始は応答イベントで、この要求をクリアします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Debug state entry is the acknowledge event that clears this request.

Abandoned accesses have the following behavior:

- an abandoned data store writes an UNKNOWN value to the target address
- an abandoned data load returns an UNKNOWN value to the register bank
- an abandoned instruction fetch returns an UNKNOWN instruction for execution
- an abandoned cache operation leaves the memory system in an UNPREDICTABLE state.

However, an abandoned access does not cause any exception.

Additional BIU requests, after Debug state has been entered, have UNPREDICTABLE behavior.

The number of ports on the processor and their protocols are implementation specific and, therefore, the detailed behavior of this bit is IMPLEMENTATION DEFINED. It is also IMPLEMENTATION DEFINED whether this behavior is supported on all ports of a processor. For example, an implementation can choose not to implement this behavior on instruction fetches.

This control bit enables the debugger to release a deadlock on the system bus so Debug state can be entered. This Debug state entry is imprecise, because the debugger only wants to know what the state of the processor was at the time the deadlock occurred. At the point where the deadlock is released, one of the following must be pending:

- a Halt request, made by also writing 1 to the Halt request bit of the DBGDRCR
- an External Debug request.

It might not be easy to infer the cause of the deadlock by reading the PC value after entering Debug state if, for example, either:

- the processor has a non-blocking cache design or a write buffer
- the deadlocked access corresponded to a load to the PC.

The effect of this bit depends on the state of the external debug interface signals:

- If the processor implements Security Extensions, a write to this bit is ignored unless **DBGEN** and **SPIDEN** are both HIGH, meaning that invasive debug is permitted in all processor states and modes.
- If the processor does not implement Security Extensions, a write to this bit is ignored unless **DBGEN** is HIGH.

For details of invasive debug authentication see Chapter C2 *Invasive Debug Authentication*.

C10.3.4 Device Power-down and Reset Control Register (DBGPRCR), v7 Debug only

The Device Power-down and Reset Control Register, DBGPRCR, controls processor functionality related to reset and power-down.

The DBGPRCR is:

- debug register 196, at offset `0x310`
- a read/write register, with more restricted access to some bits
- implemented only in v7 Debug
- when the Security Extensions are implemented, a Common register.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

破棄されたアクセスは、次の動作を行います。

- 破棄されたデータストアは、ターゲットアドレスに未知の値を書き込みます。
- 破棄されたデータロードは、レジスタバンクへ未知の値を返します。
- 破棄された命令フェッチは、未知の命令を実行用に返します。
- キャッシュ操作が破棄されると、メモリシステムは予測不能の状態になります。

ただし、破棄されたアクセスによって例外が発生することはありません。

デバッグ状態の開始後に追加の BIU 要求が行われた場合、動作は予測不能です。

プロセッサのポート番号とそのプロトコルは実装固有のため、このビットの詳細な動作は実装定義です。この動作がプロセッサのすべてのポートでサポートされるかどうかも実装定義です。例えば、実装では命令フェッチに対してこの動作を実装しないことを選択できます。

この制御ビットにより、デバッガはシステムバスのデッドロックを解放し、デバッグ状態への遷移を可能にできます。このデバッグ状態の開始は不正確です。これは、デバッガはデッドロックの発生時にプロセッサがどの状態にあったかのみを認識するためです。デッドロックが解放された時点で、次のいずれか 1 つが必ず保留中です。

- DBGDRCR のホールト要求ビットに 1 を書き込むことで行われたホールト要求
- 外部デバッグ要求

デバッグ状態の開始後に PC の値を読み出しても、デッドロックの原因を推定するのが困難な可能性があります。例えば、次のような場合が考えられます。

- プロセッサに非ブロッキングのキャッシュ設計、またはライトバッファが搭載されている。
- デッドロックの発生したアクセスが、PC へのロードに対応している。

このビットの影響は、外部デバッグインタフェース信号の状態によって異なります。

- プロセッサにセキュリティ拡張機能が実装されている場合、DBGEN と SPIDEN の両方が HIGH でない限り、このビットへの書き込みは無視されます。この両方が HIGH である場合、プロセッサのすべての状態とモードで侵襲性デバッグが許可されていることを意味します。
- プロセッサにセキュリティ拡張機能が実装されていない場合、DBGEN が HIGH でない限り、このビットへの書き込みは無視されます。

侵襲性デバッグ認証の詳細については、第 C2 章*「侵襲性デバッグの認証」*を参照して下さい。

C10.3.4 デバイス電力オフおよびリセット制御レジスタ (DBGPRCR)、v7 デバッグのみ

デバイス電力オフおよびリセット制御レジスタ (DBGPRCR) は、リセットと電力オフに関係するプロセッサの機能を制御します。

DBGPRCR の特徴は次のとおりです。

- デバッグレジスタ 196、オフセット `0x310`。
- 読み出し / 書き込みレジスタで、一部のビットはアクセスがさらに制限されています。
- v7 デバッグにのみ実装されています。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

v6 デバッグおよび v6.1 デバッグでは、レジスタ 196 は定義されていません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In v6 Debug and v6.1 Debug, register 196 is not defined.

The format of the DBGPRCR is:

Bits [31:3] Reserved, UNK/SBZP.

Hold non-debug logic reset, bit [2]

The effects of the possible values of this bit are:

0 Do not hold the non-debug logic reset on power-up or warm reset.

1 Hold the non-debug logic of the processor in reset on power-up or warm reset. The processor is held in this state until this flag is cleared to 0.

Hold non-debug logic reset is an IMPLEMENTATION DEFINED feature. If it is implemented writing 1 to this bit means the non-debug logic of the processor is held in reset after a power-up or warm reset.

Note

This bit never affects system power-up, because when implemented it resets to 0.

An external debugger can use this bit to prevent the processor running again before the debugger has had the chance to detect a power-down occurrence and restore the state of the debug registers inside the core power domain. Also, this bit can be used in conjunction with an external reset controller to take the processor into reset and hold it there while the rest of the system comes out of reset. This means a debugger can hold the processor in reset while programming other debug registers.

The effect of this bit depends on the state of the external debug interface signals:

- If the processor implements the Security Extensions, the value of this bit is ignored unless both the external debug interface signals **DBGEN** and **SPIDEN** are HIGH, meaning that invasive debug is permitted in all processor states and modes.
- If the processor does not implement the Security Extensions, the value of this bit is ignored unless **DBGEN** is HIGH.

For details of invasive debug authentication see Chapter C2 *Invasive Debug Authentication*.

If both features are supported, the bit can be written at the same time as the Warm reset request bit to force the processor into reset and hold it there, for example while programming other debug registers such as setting the Halt request bit of the DBGDRCR to take the processor into Debug state on leaving Reset. For more information, see *Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DBGPRCR のフォーマットは以下のとおりです。

ビット [31:3] 予約、UNK/SBZP

非デバッグロジック リセットの保持、ビット [2]

このビットの可能な値とその効果は次のとおりです。

0 電力オンまたはウォームリセット時に、非デバッグロジック リセットを保持しません。

1 電力オンまたはウォームリセット時に、プロセッサの非デバッグロジックをリセットに保持します。プロセッサは、このフラグが 0 にクリアされるまでこの状態に保持されます。

非デバッグロジック リセットの保持は実装定義の機能です。実装されている場合、このビットに 1 を書き込むと、電力オンまたはウォームリセットの後で、プロセッサの非デバッグロジックがリセットに保持されます。

注

このビットが実装されている場合、0 にリセットされるため、システムの電力オンには影響しません。

外部デバッガはこのビットを使用して、デバッガが電力オフの発生を検出し、コア電力ドメイン内のデバッグレジスタの状態を復元する前に、プロセッサが再度実行されることを防止できます。また、このビットを外部リセットコントローラと組み合わせて使用し、プロセッサをリセット状態にして、システムの他の部分がリセット処理を終了するまでの間、プロセッサをリセット状態に保持できます。つまり、デバッガは他のデバッグレジスタをプログラムする間、プロセッサをリセット状態に保持できます。

このビットの影響は、外部デバッグインタフェース信号の状態によって異なります。

- プロセッサにセキュリティ拡張機能が実装されている場合、外部デバッグインタフェース信号 DBGEN と SPIDEN の両方が HIGH でない限り、このビットの値は無視されます。この両方が HIGH である場合、プロセッサのすべての状態とモードで侵襲性デバッグが許可されていることを意味します。
- プロセッサにセキュリティ拡張機能が実装されていない場合、DBGEN が HIGH でない限り、このビットの値は無視されます。

侵襲性デバッグ認証の詳細については、第 C2 章*「侵襲性デバッグの認証」*を参照して下さい。

両方の機能がサポートされている場合、このビットをウォームリセット要求ビットと同時に書き込み、プロセッサを強制的にリセット状態にして、その状態に保持することができます。これは、例えば DBGDRCR のホールト要求ビットを設定し、リセットの終了時にプロセッサがデバッグ状態になるようにするなど、他のデバッグレジスタをプログラムする場合に使用できます。詳細については、*デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10- 29 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

When this bit is set to 1 the processor is not held in Debug state, and cannot enter Debug state until released from reset. While the processor is held in reset it must not accept instructions issued via the Instruction Transfer Register (DBGITR).

If Hold non-debug logic reset is not implemented this bit is RAZ/WI.

Warm reset request, bit [1]

The actions on writing to the Warm reset request bit are:

0 no action

1 request internal reset.

Warm reset request is an IMPLEMENTATION DEFINED feature. If it is implemented writing 1 to this bit issues a request for a warm reset. Typically the request is passed to an external reset controller. This means that even when a processor implements Warm reset request, whether a request causes a reset might be an IMPLEMENTATION DEFINED feature of the system that contains the processor.

Note

- This bit is always RAZ. Software must read the Sticky Reset status bit in the DBGPRSR to determine the current reset status of the processor, see *Device Power-down and Reset Status Register (DBGPRSR), v7 Debug only* on page C10-34.
- Warm reset request does not request the reset of any registers that are only reset on a debug logic reset.

The external debugger can use this bit to force the processor into reset if it does not have access to the **nRESET** input. The reset behavior is the same as warm reset driven by the **nRESET** signal. A warm reset does not cause power-down.

The effect of this bit depends on the state of the external debug interface signals:

- If the processor implements the Security Extensions, a write to this bit is ignored unless both the external debug interface signals **DBGEN** and **SPIDEN** are HIGH, meaning that invasive debug is permitted in all processor states and modes.
- If the processor does not implement the Security Extensions, a write to this bit is ignored unless **DBGEN** is HIGH.

For details of invasive debug authentication see Chapter C2 *Invasive Debug Authentication*.

Unless Hold non-debug logic reset, bit [2], is set to 1, the reset must be held only for long enough to reset the processor. The processor then leaves the reset state.

Note

If an implementation supports both features, both the Warm reset request and Hold non-debug logic reset bits can be set to 1 in a single write to the DBGPRCR. In this case the processor enters reset and is held there.

If Warm reset request is not implemented this bit is RAZ/WI.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

このビットが 1 にセットされていると、プロセッサはデバッグ状態に保持されず、リセットから解放されるまでデバッグ状態に遷移できません。リセットに保持されている間、プロセッサは命令転送レジスタ (DBGITR) 経由で発行される命令を受け付けることはできません。

非デバッグロジックリセットの保持が実装されていない場合、このビットは RAZ/WI です。

ウォームリセット要求、ビット [1]

ウォームリセット要求ビットへの書き込みは、次の動作を行います。

0 動作を行いません。

1 内部リセットを要求します。

ウォームリセット要求は実装定義の機能です。実装されている場合、このビットに 1 を書き込むと、ウォームリセット要求が発行されます。通常、この要求は外部リセットコントローラに渡されます。つまり、プロセッサにウォームリセット要求が実装されている場合でも、要求によってリセットが引き起こされるかどうかは、プロセッサを含むシステムの実装定義の機能である可能性があります。

注

- このビットは常に RAZ です。ソフトウェアは、DBGPRSR のスティッキーリセットステータス ビットを読み出して、プロセッサの現在のリセットステータスを判定する必要があります。*デバイス電力オフおよびリセットステータス レジスタ (DBGPRSR)、v7 デバッグのみ* P. C10- 34 を参照して下さい。
- ウォームリセット要求は、デバッグロジック リセットによってのみリセットされるレジスタのリセットを要求しません。

外部デバッガが nRESET 入力にアクセスできない場合、このビットを使用して、プロセッサを強制的にリセットすることができます。リセットの動作は、nRESET 信号によって駆動されるウォームリセットと同じです。ウォームリセットは電力オフを引き起こしません。

このビットの影響は、外部デバッグインタフェース信号の状態によって異なります。

- プロセッサにセキュリティ拡張機能が実装されている場合、外部デバッグインタフェース信号 DBGEN と SPIDEN の両方が HIGH でない限り、このビットへの書き込みは無視されます。この両方が HIGH である場合、プロセッサのすべての状態とモードで侵襲性デバッグが許可されていることを意味します。
- プロセッサにセキュリティ拡張機能が実装されていない場合、DBGEN が HIGH でない限り、このビットへの書き込みは無視されます。

侵襲性デバッグ認証の詳細については、第 C2 章*「侵襲性デバッグの認証」*を参照して下さい。

非デバッグロジック リセットの保持（ビット [2]）が 1 にセットされていない限り、リセットはプロセッサのリセットに必要な時間だけ保持される必要があります。その後で、プロセッサはリセット状態を終了します。

注

実装で両方の機能がサポートされている場合、ウォームリセット要求ビットと非デバッグロジック リセット保持ビットの両方を、DBGPRCR への単一の書き込みで 1 にセットできます。この場合、プロセッサはリセット状態に遷移し、その状態に保持されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DBGnoPWRDWN, bit [0]

No power-down bit, DBGnoPWRDWN. This bit controls the **DBGNOPWRDWN** signal, if it is implemented. The possible values of this bit are:

0 drive **DBGNOPWRDWN** LOW

1 drive **DBGNOPWRDWN** HIGH.

DBGNOPWRDWN is an IMPLEMENTATION DEFINED feature. If it is implemented, setting this bit drives the **DBGNOPWRDWN** signal HIGH, requesting the power controller to work in an emulation mode where the processor is not actually powered down when requested. For more information, see *DBGNOPWRDWN* on page AppxA-9.

If the **DBGNOPWRDWN** signal is not implemented this bit is RAZ/WI.

C10.3.5 Device Power-down and Reset Status Register (DBGPRSR), v7 Debug only

The Device Power-down and Reset Status Register, DBGPRSR, holds information about the reset and power-down state of the processor.

The DBGPRSR is:

- debug register 197, at offset `0x314`
- a read-only register, with reads of the register also resetting some register bits
- implemented only in v7 Debug
- when the Security Extensions are implemented, a Common register.

In v6 Debug and v6.1 Debug, register 197 is not defined.

The format of the DBGPRSR is:

Bits [31:4] Reserved, UNK.

Sticky Reset status, bit [3]

The meanings of the Sticky Reset status bit values are:

0 the non-debug logic of the processor has not been reset since the last time this register was read

1 the non-debug logic of the processor has been reset since the last time this register was read.

This bit is cleared to 0 on a read of the DBGPRSR when the non-debug logic of the processor is not in reset state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ウォームリセット要求が実装されていない場合、このビットは RAZ/WI です。

DBGnoPWRDWN、ビット [0]

電力オフなしビット、DBGnoPWRDWN。このビットは、DBGNOPWRDWN 信号が実装されている場合に、制御を行います。このビットの可能な値は次のとおりです。

0 DBGNOPWRDWN を LOW に駆動します。

1 DBGNOPWRDWN を HIGH に駆動します。

DBGNOPWRDWN は実装定義の機能です。実装されている場合、このビットをセットすると DBGNOPWRDWN 信号が HIGH に駆動され、電力コントローラにエミュレーションモードでの動作を要求します。このモードでは、電力オフが要求されたとき、プロセッサが実際には電力オフされません。詳細については、*DBGNOPWRDWN* P. 付録 A- 9 を参照して下さい。

DBGNOPWRDWN 信号が実装されていない場合、このビットは RAZ/WI です。

C10.3.5 デバイス電力オフおよびリセットステータス レジスタ (DBGPRSR)、v7 デバッグのみ

デバイス電力オフおよびリセットステータス レジスタ (DBGPRSR) は、プロセッサのリセットと電力オフ状態についての情報を保持します。

DBGPRSR の特徴は次のとおりです。

- デバッグレジスタ 197、オフセット `0x314`。
- 読み出し専用レジスタで、レジスタの読み出しにより、一部のビットが同時にリセットされます。
- v7 デバッグにのみ実装されています。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

v6 デバッグおよび v6.1 デバッグでは、レジスタ 197 は定義されていません。

DBGPRSR のフォーマットは次のとおりです。

ビット [31:4] 予約、UNK

スティッキーリセット ステータス、ビット [3]

スティッキーリセットステータス ビットの値の意味は次のとおりです。

0 プロセッサの非デバッグロジックは、このレジスタが最後に読み出された後にリセットされていません。

1 プロセッサの非デバッグロジックは、このレジスタが最後に読み出された後にリセットされました。

このビットは、プロセッサの非デバッグロジックがリセット状態でないときに DBGPRSR を読み出すと、0 にクリアされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When the non-debug logic of the processor is in reset state, the Sticky Reset status bit is set to 1.

Reads of DBGPRSR made when the non-debug logic of the processor is in reset state return 1 for Sticky Reset status and do not change the value of Sticky Reset status.

Reads of DBGPRSR made when the non-debug logic of the processor is not in reset state return the current value of Sticky Reset status, and then clear Sticky Reset status to 0.

Note

- *Reset state* is defined in *Reset state* on page C10-37.
- On a read access, the Sticky Reset status bit can be cleared only as a side effect of the read. When a read is made through the memory-mapped interface with the Software Lock set, side-effects are not permitted, and therefore the bit is not cleared. For more information, see *Permission summaries for memory-mapped and external debug interfaces* on page C6-45.
- Bits [3:2] of DBGPRSR never read as 0b01.

The debug logic reset value for the Sticky Reset status bit is UNKNOWN.

Reset status, bit [2]

The meanings of the Reset status bit values are:

0 the non-debug logic of the processor is not currently held in reset state

1 the non-debug logic of the processor is currently held in reset state.

Note

Reset state is defined in *Reset state* on page C10-37.

Reads of the DBGPRSR made when the non-debug logic of the processor is in reset state return 1 for the Reset status.

Reads of the DBGPRSR made when the non-debug logic of the processor is not in reset state return 0 for the Reset status.

Sticky Power-down status, bit [1]

The meanings of the Sticky Power-down status bit values are:

0 the processor has not powered down since the last time this register was read

1 the processor has powered down since the last time this register was read.

This bit is cleared to 0 on a read of the DBGPRSR when the processor is in the powered-up state.

Note

If the implementation supports separate core and debug power domains, the Sticky Power-down status bit reflects the state of the core power domain. Powered-up and powered-down are defined in *Powered-up state* on page C10-37.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

プロセッサの非デバッグロジックがリセット状態のとき、スティッキーリセット ステータス ビットは 1 にセットされます。

プロセッサの非デバッグロジックがリセット状態のときに DBGPRSR を読み出すと、スティッキーリセット ステータスの値として 1 が返され、スティッキーリセット ステータスの値は変更されません。

プロセッサの非デバッグロジックがリセット状態でないときに DBGPRSR を読み出すと、スティッキーリセット ステータスの現在の値が返され、その後でスティッキーリセット ステータスが 0 にクリアされます。

注

- *リセット状態*の定義については、*リセット時の状態* P. C10-37 を参照して下さい。
- 読み出しアクセス時に、スティッキーリセットステータス ビットは、読み出しの副作用としてのみクリアできます。ソフトウェアロックがセットされた状態で、メモリマップされたインタフェース経由で読み出しを行うと、副作用は許可されないため、このビットはクリアされません。詳細については、*メモリマップおよび外部デバッグインタフェースのアクセス許可の概要* P. C6-45 を参照して下さい。
- DBGPRSR のビット [3:2] が 0b01 として読み出されることはありません。

スティッキーリセットステータス ビットのデバッグロジック リセット値は未知です。

リセットステータス、ビット [2]

リセットステータス ビットの値の意味は次のとおりです。

0 プロセッサの非デバッグロジックは、現在リセット状態に保持されていません。

1 プロセッサの非デバッグロジックは、現在リセット状態に保持されています。

注

*リセット状態*の定義については、*リセット時の状態* P. C10-37 を参照して下さい。

プロセッサの非デバッグロジックがリセット状態のときに DBGPRSR を読み出すと、リセット状態として 1 が返されます。

プロセッサの非デバッグロジックがリセット状態でないときに DBGPRSR を読み出すと、リセット状態として 0 が返されます。

スティッキー電力オフステータス、ビット [1]

スティッキー電力オフステータス ビットの値の意味は次のとおりです。

0 このレジスタが最後に読み出された後で、プロセッサの電力がオフになっていません。

1 このレジスタが最後に読み出された後で、プロセッサの電力がオフになりました。

プロセッサが電力オン状態のときに DBGPRSR を読み出すと、このビットは 0 にクリアされます。

注

実装で、分離したコアおよびデバッグ電力ドメインがサポートされている場合、スティッキー電力オフステータス ビットは、コア電力ドメインの状態を反映しています。電力オンと電力オフの定義については、*電力オン状態* P. C10-37 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When the processor is in the powered-down state, the Sticky Power-down status bit is set to 1.

Reads of DBGPRSR made when the processor is in the powered down state return 1 for Sticky Power-down status and do not change the value of Sticky Power-down status.

Reads of DBGPRSR made when the processor is in the powered-up state return the current value of Sticky Power-down status, and then clear Sticky Power-down status to 0.

Note

- The value 0b00 for DBGPRSR[1:0], indicating certain of the debug registers cannot be accessed but have not lost their value, is not permitted in v7 Debug.
- On a read access, the Sticky Power-down status bit can be cleared only as a side effect of the read. When a read is made through the memory-mapped interface with the Software Lock set, side-effects are not permitted, and therefore the bit is not cleared. For more information, see *Permission summaries for memory-mapped and external debug interfaces* on page C6-45.

If this bit is set to 1, accesses to certain registers return an error response. For more information, see *Permissions in relation to power-down* on page C6-28.

The debug logic reset value for the Sticky Power-down status bit is UNKNOWN.

Power-up status, bit [0]

The meanings of the Power-up status bit values are:

0 The processor is powered-down. Certain debug registers cannot be accessed.

1 The processor is powered-up. All debug registers can be accessed.

Note

If the implementation supports separate core and debug power domains, the Power-up status bit reflects the state of the core power domain. Powered-up and powered-down are defined in *Powered-up state* on page C10-37.

The Power-up status bit reads the value of the **DBGPWRDUP** input on the external debug interface. For details of the **DBGPWRDUP** input see *DBGPWRDUP* on page AppxA-10.

Reads of DBGPRSR made when the processor is in the powered up state return 1 for Power-up status.

Reads of DBGPRSR made when the processor is in the powered down state return 0 for Power-up status.

For more information, see *Power domains and debug* on page C6-5.

Note

If only a single power-domain is implemented:

- bit [0] of the DBGPRSR is RAO
- bit [1] of the DBGPRSR can be implemented as RAZ.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

プロセッサが電力オフ状態のとき、スティッキー電力オフステータス ビットは 1 にセットされています。

プロセッサが電力オフ状態のときに DBGPRSR を読み出すと、スティッキー電力オフステータスの値として 1 が返され、スティッキー電力オフステータスの値は変更されません。

プロセッサが電力オン状態のときに DBGPRSR を読み出すと、スティッキー電力オフステータスの現在の値が返され、その後でスティッキー電力オフステータスは 0 にクリアされます。

注

- DBGPRSR[1:0] の値 0b00 は、特定のデバッグレジスタにアクセスできないが、それらのレジスタの値が消失していないことを意味しますが、この値は v7 デバッグでは許可されません。
- 読み出しアクセス時に、スティッキー電力オフステータス ビットは、読み出しの副作用としてのみクリアできます。ソフトウェアロックがセットされた状態で、メモリマップされたインタフェース経由で読み出しを行うと、副作用は許可されないため、このビットはクリアされません。詳細については、*メモリマップおよび外部デバッグインタフェースのアクセス許可の概要* P. C6- 45 を参照して下さい。

このビットが 1 にセットされている場合、特定のレジスタへのアクセスでエラー応答が返されます。詳細については、*電力オフとアクセス許可との関係* P. C6- 27 を参照して下さい。

スティッキー電力オフステータス ビットのデバッグロジック リセット値は未知です。

電力オンステータス、ビット [0]

電力オンステータス ビットの値の意味は次のとおりです。

0 プロセッサの電力はオフです。特定のデバッグレジスタにアクセスできません。

1 プロセッサの電力はオンです。すべてのデバッグレジスタにアクセスできます。

注

実装で、分離したコアおよびデバッグ電力ドメインがサポートされている場合、電力オンステータス ビットは、コア電力ドメインの状態を反映しています。電力オンと電力オフの定義については、*電力オン状態* P. C10- 37 を参照して下さい。

電力オンステータス ビットの読み出し値は、外部デバッグインタフェースの DBGPWRDUP 入力の値です。DBGPWRDUP 入力の詳細については、*DBGPWRDUP* P. 付録 A- 10 を参照して下さい。

プロセッサが電力オン状態のときに DBGPRSR を読み出すと、電力オンステータスとして 1 が返されます。

プロセッサが電力オフ状態のときに DBGPRSR を読み出すと、電力オンステータスとして 0 が返されます。

詳細については、*電力ドメインとデバッグ* P. C6- 5 を参照して下さい。

注

単一の電力ドメインのみが実装されている場合、ビットの値は次のようになります。

- DBGPRSR のビット [0] は RAO です。
- DBGPRSR のビット [1] は RAZ として実装できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Reset state

In the reset scheme described in *Recommended reset scheme for v7 Debug* on page C6-16, the non-debug logic of the processor enters reset state following the assertion of at least one of:

- the internal or warm reset input, **nRESET**
- The power-up reset inputs, **nCOREPORESET** and **nSYSPORESET**.

All of these reset signals are asserted LOW.

Also, writing 1 to the Warm reset request bit of the DBGPRCR might cause the non-debug logic of the processor to enter reset state, see *Device Power-down and Reset Control Register (DBGPRCR), v7 Debug only* on page C10-31.

The processor stops executing instructions before it enters reset state.

The non-debug logic of the processor remains in reset state until:

- all of the reset signals **nRESET**, **nCOREPORESET**, and **nSYSPORESET**, are deasserted HIGH
- the Hold warm reset bit in the Device Power-down and Reset Control Register (DBGPRCR) is 0.

Note

One effect of asserting **nSYSPORESET** LOW is to place the debug logic into a reset state. In this state the DBGPRSR is not accessible.

The processor then resumes execution of instructions with the Reset exception.

Powered-up state

The processor is in the powered-up state when **DBGPWRDUP** is HIGH, and is in the powered-down state when **DBGPWRDUP** is LOW. Changing from powered-down state to powered-up state requires a reset of the processor.

If the implementation supports separate core and debug power domains, powered-up and powered-down state refer to the state of the core power domain.

Powered-up status is not affected by the reset state of the processor, whether that reset is:

- a power-up reset, **nCOREPORESET** or **nSYSPORESET**
- a warm reset, **nRESET**
- a reset occurring because the Hold non-debug logic reset bit in the Device Power-down and Reset Control Register (DBGPRCR) is set to 1.

For more information, see *Reset and power-down support* on page C6-4.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

リセット時の状態

v7 デバッグで推奨されるリセット方式 P. C6-16 に説明されているリセット方式では、次のうち最低 1 つがアサートされると、プロセッサの非デバッグロジックがリセット状態に遷移します。

- 内部またはウォームリセット入力、nRESET
- 電力オンリセット入力、nCOREPORESET および nSYSPORESET

これらのリセット信号はすべて、LOW にアサートされます。

また、DBGPRCR のウォームリセット要求ビットに 1 を書き込むと、プロセッサの非デバッグロジックがリセット状態に遷移することがあります。*デバイス電力オフおよびリセット制御レジスタ (DBGPRCR)、v7 デバッグのみ* P. C10-31 を参照して下さい。

プロセッサは、リセット状態に遷移する前に、命令の実行を停止します。

プロセッサの非デバッグロジックは、次の条件が満たされるまでリセット状態に保持されます。

- nRESET、nCOREPORESET、nSYSPORESET のすべてのリセット信号が、HIGH にアサート解除される。
- デバイス電力オフおよびリセット制御レジスタ (DBGPRCR) のウォームリセット保持ビットが 0 になる。

注

nSYSPORESET を LOW にアサートすることの影響の 1 つは、デバッグロジックがリセット状態に置かれることです。この状態では、DBGPRSR にアクセスできません。

次に、プロセッサはリセット例外で命令の実行を再開します。

電力オン状態

プロセッサは、DBGPWRDUP が HIGH のときに電力オン状態で、DBGPWRDUP が LOW のときに電力オフ状態です。電力オフ状態から電力オン状態に変更するには、プロセッサのリセットが必要です。

実装で、分離したコアおよびデバッグ電力ドメインがサポートされている場合、電力オンと電力オフの状態は、コア電力ドメインの状態を指します。

電力オンの状態は、そのリセットが次のどれであるかにかかわらず、プロセッサのリセット状態の影響を受けません。

- 電力オンリセット、nCOREPORESET または nSYSPORESET
- ウォームリセット、nRESET
- デバイス電力オフおよびリセット制御レジスタ (DBGPRCR) の非デバッグロジック リセット保持ビットが 1 にセットされたために発生したリセット

詳細については、*リセットと電力オフのサポート* P. C6-4 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.3.6 Program Counter Sampling Register (DBGPCSR)

The Program Counter Sampling Register, DBGPCSR, enables a debugger to sample the Program Counter (PC).

DBGPCSR is defined only in the v7 Debug architecture. However, an ARMv6 implementation might implement DBGPCSR as part of the external debug interface.

In v7 Debug:

- It is IMPLEMENTATION DEFINED whether DBGPCSR is:
  - — not implemented
  - — implemented as debug register 33, at offset `0x084`
  - — implemented as debug register 40, at offset `0x0A0`
  - — implemented both as debug register 33 and as debug register 40.
- When DBGPCSR is implemented both as debug register 33 and as debug register 40, the two register numbers are aliases of each other.
- You can determine whether, or how, DBGPCSR is implemented as follows:
  - — If DBGDIDR.PCSR_imp is 1, DBGPCSR is implemented as debug register 33. Otherwise, reads of register 33 return an UNKNOWN value.
  - — If DBGDIDR.DEVID_imp is 1 and DBGDEVID.PCsample is non-zero, DBGPCSR is implemented as debug register 40. Otherwise, debug register 40 is reserved.

When implemented, the DBGPCSR is:

- a read-only register
- when the Security Extensions are implemented, a Common register.

Any read through the Extended CP14 interface of a CP14 register that maps to the DBGPCSR is UNDEFINED in User mode and UNPREDICTABLE in privileged modes.

ARM deprecates reading a PC sample through register 33 when the DBGPCSR is also implemented as register 40.

The format of the DBGPCSR is:

Program Counter Sample value, bits [31:2]

The sampled value of bits [31:2] of the PC. The sampled value is an instruction address plus an offset that depends on the processor instruction set state. See *Memory addresses* on page C3-23 for a definition of the *Instruction Virtual Address* (IVA) read through the DBGPCSR.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.3.6 プログラムカウンタサンプリング レジスタ (DBGPCSR)

プログラムカウンタサンプリング レジスタ (DBGPCSR) は、デバッガがプログラムカウンタ (PC) のサンプリングを行うために使用されます。

DBGPCSR は、v7 デバッグアーキテクチャでのみ定義されています。ただし、ARMv6 実装に外部デバッグインタフェースの一部として DBGPCSR が実装されていることがあります。

v7 デバッグでは、次の規則が適用されます。

- DBGPCSR は、実装定義により次のいずれかです。
  - – 実装されていない。
  - – デバッグレジスタ 33、オフセット `0x084` として実装されている。
  - – デバッグレジスタ 40、オフセット `0x0A0` として実装されている。
  - – デバッグレジスタ 33 とデバッグレジスタ 40 の両方として実装されている。
- DBGPCSR がデバッグレジスタ 33 とデバッグレジスタ 40 の両方として実装されている場合、これら 2 つのレジスタ番号はお互いのエイリアスです。
- DBGPCSR が実装されているか、実装されている場合にはどのように実装されているかは、次のように判定できます。
  - – DBGDIDR.PCSR_imp が 1 の場合、DBGPCSR はデバッグレジスタ 33 として実装されています。それ以外の場合、レジスタ 33 の読み出しで返される値は未知です。
  - – DBGDIDR.DEVID_imp が 1 で、DBGDEVID.PCsample が 0 以外の場合、DBGPCSR はデバッグレジスタ 40 として実装されています。それ以外の場合、デバッグレジスタ 40 は予約されています。

実装されている場合、DBGPCSR の特徴は次のとおりです。

- 読み出し専用レジスタ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

CP14 レジスタの拡張 CP14 インタフェースを経由し、DBGPCSR にマッピングされるすべての読み出しは、ユーザモードでは未定義で、特権モードでは予測不能です。

DBGPCSR がレジスタ 40 としても実装されている場合、レジスタ 33 経由で PC サンプルを読み出すことは推奨されません。

DBGPCSR のフォーマットは次のとおりです。

プログラムカウンタ サンプルの値、ビット [31:2]

PC のビット [31:2] のサンプリングされた値。サンプリングされた値は、命令のアドレスに、プロセッサの命令セット状態によって異なるオフセットを加えた値です。DBGPCSR 経由で読み出される *命令仮想アドレス* (IVA) の定義については、*メモリアドレス* P. C3-23 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Meaning of PC Sample Value, bits [1:0]

The permitted values of this field are:

0b00 ((DBGPCSR[31:2] << 2) - 8) references an ARM state instruction

0bx1 ((DBGPCSR[31:1] << 1) - 4) references a Thumb or ThumbEE state instruction

0b10 IMPLEMENTATION DEFINED.

This field encodes the processor instruction set state, so that the profiling tool can calculate the true instruction address by subtracting the appropriate offset from the value sampled in bits [31:2] of the register.

For more information about Program Counter sampling, see *Program Counter sampling* on page C8-2.

C10.3.7 Context ID Sampling Register (DBGCIDSR)

The Context ID Sampling Register, DBGCIDSR, samples the CONTEXTIDR whenever the Program Counter Sampling Register, DBGPCSR, samples the Program Counter. This enables a debugger to associate a Program Counter sample with the process running on the processor.

DBGCIDSR is defined only in the v7 Debug architecture. However, an ARMv6 implementation might implement DBGCIDSR as part of the external debug interface.

In v7 Debug:

- It is IMPLEMENTATION DEFINED whether DBGCIDSR is implemented.
- If DBGDIDR.DEVID_imp is 1 and DBGDEVID.PCsample is 0b0010, DBGCIDSR is implemented as debug register 41. Otherwise, debug register 41 is reserved.

When implemented, the DBGCIDSR is:

- debug register 41, at offset `0x0A4`
- a read-only register
- when the Security Extensions are implemented, a Common register.

Any read through the Extended CP14 interface of the CP14 register that maps to the DBGCIDSR is UNDEFINED in User mode and UNPREDICTABLE in privileged modes.

The format of the DBGCIDSR is:

31 … 0
CONTEXTIDR sample value

CONTEXTIDR sample value, bits [31:0]

The value of the Context ID Register, CONTEXTIDR, associated with the last PC sample read from DBGPCSR.

The core logic reset value of the DBGCIDSR is UNKNOWN.

For more information about Program Counter sampling, see *Program Counter sampling* on page C8-2.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

PC サンプリング値の意味、ビット [1:0]

このフィールドに許容される値は次のとおりです。

0b00 ((DBGPCSR[31:2] << 2) − 8) が ARM 状態の命令を参照します。

0bx1 ((DBGPCSR[31:1] << 1) − 4) が Thumb または ThumbEE 状態の命令を参照します。

0b10 実装定義

このフィールドには、プロセッサの命令セット状態がエンコードされています。プロファイリングツールは、この命令セット状態を確認したうえで、レジスタのビット [31:2] にサンプリングされた値から適切なオフセットを差し引くことで、実際の命令アドレスを計算できます。

プログラムカウンタ サンプリングの詳細については、*プログラムカウンタのサンプリング* P. C8-2 を参照して下さい。

C10.3.7 コンテキスト ID サンプリングレジスタ (DBGCIDSR)

コンテキスト ID サンプリングレジスタ (DBGCIDSR) は、プログラムカウンタサンプリング レジスタ (DBGPCSR) がプログラムカウンタをサンプリングするたびに、CONTEXTIDR をサンプリングします。これによって、デバッガは、プログラムカウンタのサンプルを、プロセッサで実行されているプロセスと関連付けることができます。

DBGCIDSR は、v7 デバッグアーキテクチャでのみ定義されています。ただし、ARMv6 実装に外部デバッグインタフェースの一部として DBGCIDSR が実装されていることがあります。

v7 デバッグでは、次の規則が適用されます。

- DBGCIDSR が実装されるかどうかは実装定義です。
- DBGDIDR.DEVID_imp が 1 で、DBGDEVID.PCsample が 0b0010 の場合、DBGCIDSR はデバッグレジスタ 41 として実装されています。それ以外の場合、デバッグレジスタ 41 は予約されています。

実装されている場合、DBGCIDSR の特徴は次のとおりです。

- デバッグレジスタ 41、オフセット `0x0A4`
- 読み出し専用レジスタ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

CP14 レジスタの拡張 CP14 インタフェースを経由し、DBGCIDSR にマッピングされるすべての読み出しは、ユーザモードでは未定義で、特権モードでは予測不能です。

DBGCIDSR のフォーマットは次のとおりです。

31 … 0
CONTEXTIDRサンプルの値

CONTEXTIDR サンプルの値、ビット [31:0]

コンテキスト ID レジスタ (CONTEXTIDR) の値は、DBGPCSR から最後に読み出された PC サンプルに関連付けられています。

コアロジック リセット時の DBGCIDSR の値は未知です。

プログラムカウンタ サンプリングの詳細については、*プログラムカウンタのサンプリング* P. C8-2 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.4 Instruction and data transfer registers

This section describes the registers that are used to transfer data between an external debugger and the ARM processor. It contains the following subsections:

- *Host to Target Data Transfer Register (DBGDTRRX)*
- *Target to Host Data Transfer Register (DBGDTRTX)* on page C10-43
- *Instruction Transfer Register (DBGITR)* on page C10-46.

The following registers and flags form the Debug Communications Channel:

- the DBGDTRRX Register, see *Host to Target Data Transfer Register (DBGDTRRX)*
- the DBGDTRTX Register, see *Target to Host Data Transfer Register (DBGDTRTX)* on page C10-43
- the RXfull, TXfull, TXfull_l, and RXfull_l flags in the DBGDSCR, see:
  - — the flag descriptions in *Debug Status and Control Register (DBGDSCR)* on page C10-10
  - — *Access controls on the external view of the DCC registers and DBGITR, v7 Debug only* on page C10-21.

C10.4.1 Host to Target Data Transfer Register (DBGDTRRX)

The Host to Target Data Transfer Register, DBGDTRRX, is used by an external host to transfer data to the ARM processor. For example it is used by a debugger transferring commands and data to a debug target.

The DBGDTRRX Register is:

- Debug register 32, at offset `0x080`.
- A component of the *Debug Communication Channel* (DCC).
- Accessed through two views:
  - — DBGDTRRXint, the internal view
  - — DBGDTRRXext, the external view.

  See *Internal and external views of the DBGDSCR and the DCC registers* on page C6-21 for definitions of the internal and external views.
- When the Security Extensions are implemented, a Common register.

The behavior of accesses to the DBGDTRRX Register depends on:

- which view is being accessed
- the values of flags in the DCC.

For more information, see *Access to the DBGDTRRX Register* on page C10-41.

The architectural status of the DBGDTRRX Register depends on the Debug architecture version:

ARMv6 DBGDTRRX was previously named rDTR. DBGDTRRXext is not defined in ARMv6. However, the DBGDTRRXext functionality must be implemented as part of the external debug interface.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C10.4 命令およびデータ転送レジスタ

このセクションでは、外部デバッガと ARM プロセッサとの間でデータを転送するために使用されるレジスタについて説明します。このセクションは、次のサブセクションで構成されています。

- *ホストからターゲットへのデータ転送レジスタ (DBGDTRRX)*
- *ターゲットからホストへのデータ転送レジスタ (DBGDTRTX)* P. C10-43
- *命令転送レジスタ (DBGITR)* P. C10-46

デバッグ通信チャネルは、次のレジスタとフラグで構成されています。

- DBGDTRRX レジスタ。*ホストからターゲットへのデータ転送レジスタ (DBGDTRRX)* を参照して下さい。
- DBGDTRTX レジスタ。*ターゲットからホストへのデータ転送レジスタ (DBGDTRTX)* P. C10-43 を参照して下さい。
- DBGDSCR の RXfull、TXfull、TXfull_l、RXfull_l フラグ。次のセクションを参照して下さい。
  - – *デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 のフラグの説明
  - – *DCC レジスタと DBGITR の外部ビューへのアクセス制御、v7 デバッグのみ* P. C10-21

C10.4.1 ホストからターゲットへのデータ転送レジスタ (DBGDTRRX)

ホストからターゲットへのデータ転送レジスタ (DBGDTRRX) は、外部ホストが ARM プロセッサへデータを転送するために使用されます。例えば、デバッガがコマンドとデータをデバッグターゲットへ転送するために使用されます。

DBGDTRRX レジスタには、次の特徴があります。

- デバッグレジスタ 32、オフセット `0x080`
- *デバッグ通信チャネル* (DCC) のコンポーネント
- 2 つのビューを経由してアクセスされます。
  - – DBGDTRRXint、内部ビュー
  - – DBGDTRRXext、外部ビュー

  内部ビューと外部ビューの定義については、*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6-21 を参照して下さい。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

DBGDTRRX レジスタへのアクセスの動作は、次の要因に依存します。

- アクセスされるビュー
- DCC のフラグの値

詳細については、*DBGDTRRX レジスタへのアクセス* P. C10-41 を参照して下さい。

DBGDTRRX レジスタのアーキテクチャ的なステータスは、デバッグアーキテクチャのバージョンによって異なります。

ARMv6 DBGDTRRX は、以前は rDTR という名前でした。ARMv6 では DBGDTRRXext は定義されていません。ただし、外部デバッグインタフェースの一部として DBGDTRRXext の機能を実装する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

v7 Debug The Extended CP14 interface instructions that access DBGDTRRXext, if implemented, are UNPREDICTABLE in Debug state. For more information, see *Internal and external views of the DBGDSCR and the DCC registers* on page C6-21 and *Extended CP14 interface* on page C6-33.

The format of the DBGDTRRX Register is:

31 … 0
Host to target data

Host to target data, bits [31:0]

One word of data for transfer from the debug host to the debug target.

The debug logic reset value of the DBGDTRRX Register is UNKNOWN.

Access to the DBGDTRRX Register

The behavior on various accesses to the DBGDTRRX Register is described in the following tables:

- Table C10-4 shows the behavior of accesses to DBGDTRRXint
- Table C10-5 on page C10-42 shows the behavior of read accesses to DBGDTRRXext
- Table C10-6 on page C10-42 shows the behavior of write accesses to DBGDTRRXext.

To access the DBGDTRRXint Register you read the CP14 registers using either:

- an MRC instruction with <opc1> set to 0, <CRn> set to c0, <CRm> set to c5, and <opc2> set to 0
- an STC instruction with <CRd> set to c5.

Both instructions read only one word from the DBGDTRRXint Register. For example:

```
MCR p14,0,<Rd>,c0,c5,0 ; Read DBGDTRRXint Register
STC p14,c5,[<Rn>],#4   ; Read a word from the DBGDTRRXint Register and write it to memory
```

Table C10-4 Behavior of accesses to DBGDTRRXint

Access	RXfull	Action	New RXfull
Read	0	Returns an UNKNOWN value.	Unchanged
	1	Returns DBGDTRRX contents	0
Write	X	Not possible. There is no operation that writes to DBGDTRRXint	-

Note

- If the STC instruction that reads DBGDTRRXint aborts, the contents of DBGDTRRX and the value of the RXfull flag are UNKNOWN.
- The behavior on accesses to DBGDTRRXint does not depend on the value of RXfull_l,

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

v7 デバッグ DBGDTRRXext にアクセスする拡張 CP14 インタフェース命令は、それが実装されている場合、デバッグ状態では予測不能です。詳細については、*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6-21 と *拡張 CP14 インタフェース* P. C6-33 を参照して下さい。

DBGDTRRX レジスタのフォーマットは次のとおりです。

31 … 0
ホストからターゲットへのデータ

ホストからターゲットへのデータ、ビット [31:0]

デバッグホストからデバッグターゲットへ転送される 1 ワードのデータ。

デバッグロジック リセット時の DBGDTRRX レジスタの値は未知です。

DBGDTRRX レジスタへのアクセス

DBGDTRRX レジスタに対する各種のアクセスの動作については、次の表を参照して下さい。

- DBGDTRRXint へのアクセスの動作を、表 C10-4 に示します。
- DBGDTRRXext への読み出しアクセスの動作を、P. C10-42 の表 C10-5 に示します。
- P. C10-42 の表 C10-6 DBGDTRRXext への書き込みアクセスの動作を、に示します。

DBGDTRRXint レジスタにアクセスするには、次のいずれかの命令を使用して CP14 レジスタを読み出します。

- MRC 命令で <opc1> を 0、CRn<> を c0、<CRm> を c5、<opc2> を 0 に設定する。
- STC 命令で <CRd> を c5 に設定する。

どちらの命令も、DBGDTRRXint レジスタから 1 ワードのみを読み出します。次に例を示します。

```
MCR p14,0,<Rd>,c0,c5,0 ; Read DBGDTRRXint Register
STC p14,c5,[<Rn>],#4   ; Read a word from the DBGDTRRXint Register and write it to memory
```

表 C10-4 DBGDTRRXint へのアクセスの動作

アクセス	RXfull	動作	新しい RXfull
読み出し	0	未知 (UNKNOWN) の値が返されます。	変更なし
	1	DBGDTRRX の内容が返されます。	0
書き込み	X	不可能。DBGDTRRXint への書込み操作は存在しません。	–

注

- DBGDTRRXint を読み出す STC 命令がアボートした場合、DBGDTRRX の内容と RXfull フラグの値は未知です。
- DBGDTRRXint へのアクセス時の動作は、RXfull_l の値に依存しません。
- DBGDTRRXint へのアクセスでは、RXfull_l の値は更新されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- Accesses to DBGDTRRXint do not update the value of RXfull_l.

Accesses to DBGDTRRXext can be made through:
- the Extended CP14 interface, if implemented
- the memory-mapped interface, if implemented
- the external debug interface.

Table C10-5 Behavior of read accesses to DBGDTRRXext

Access mode[a]	Flag[b]	Flag value	Action	New RXfull	New RXfull_l
X	RXfull	0	Returns an UNKNOWN value	Unchanged	Unchanged
		1	Returns DBGDTRRX contents	Unchanged	Unchanged

a. For more information, see *Access controls on the external view of the DCC registers and DBGITR, v7 Debug only* on page C10-21.
b. This column indicates which of the RXfull, RXfull_l and InstrCompl flags are used to control the access. The access does not depend on the value of any other flags.

Table C10-6 Behavior of write accesses to DBGDTRRXext

Access mode[a]	Flag[b]	Flag value	Action	New RXfull	New RXfull_l
Non-blocking	RXfull_l	0	Writes to DBGDTRRX[c]	1[c]	1[c]
		1	Write is ignored.	Unchanged	Unchanged
Stall	RXfull	0	Writes to DBGDTRRX[c]	1[c]	1[c]
		1	Stalls[c] until (RXfull = 0)	-	-
Fast	InstrCompl	0	Stalls[c] until (InstrCompl = 1)	-	-
		1	Writes to DBGDTRRX[c, d] and issues the instruction from the DBGITR[c, e]	1[c, d, e]	1[c, d, e]

a. For more information, see *Access controls on the external view of the DCC registers and DBGITR, v7 Debug only* on page C10-21.
b. This column indicates which of the RXfull, RXfull_l and InstrCompl flags are used to control the access. The access does not depend on the value of any other flags.
c. If the write is made through the memory-mapped interface and the Software Lock is set, the registers are read-only and accesses have no side-effects. This means that:
   DBGDTRRX, RXfull, RXfull_l, InstrCompl and InstrCompl_l are unchanged
   the access completes immediately
   in Fast mode no instruction is issued.
   For more information, see *Permission summaries for memory-mapped and external debug interfaces* on page C6-45.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

DBGDTRRXext へのアクセスは、次のいずれかを経由して行うことができます。
- 拡張 CP14 インタフェース（実装されている場合）
- メモリマップされたインタフェース（実装されている場合）
- 外部デバッグインタフェース

表 C10-5 DBGDTRRXext への読み出しアクセスの動作

アクセスモード[a]	フラグ[b]	フラグの値	動作	新しい RXfull	新しい RXfull_l
X	RXfull	0	未知 (UNKNOWN) の値が返されます。	変更なし	変更なし
		1	DBGDTRRX の内容が返されます。	変更なし	変更なし

a. 詳細については、*DCC レジスタと DBGITR の外部ビューへのアクセス制御、v7 デバッグのみ* P. C10-21 を参照して下さい。
b. この列は、アクセスの制御に RXfull、RXfull_l、InstrCompl フラグのどれが使用されるかを示しています。アクセスは他のフラグの値には依存しません。

表 C10-6 DBGDTRRXext への書き込みアクセスの動作

アクセスモード[a]	フラグ[b]	フラグの値	動作	新しい RXfull	新しい RXfull_l
非ブロッキング	RXfull_l	0	DBGDTRRX へ書き込みを行います。[c]	1[c]	1[c]
		1	書き込みは無視されます。	変更なし	変更なし
ストール	RXfull	0	DBGDTRRX へ書き込みを行います。[c]	1[c]	1[c]
		1	(RXfull = 0) になるまでストールします。[c]	–	–
高速	InstrCompl	0	(InstrCompl = 1) になるまでストールします。[c]	–	–
		1	DBGDTRRX へ書き込みを行い[c.d] DBGITR から命令を発行します。[c.e]	1[c, d, e]	1[c, d, e]

a. 詳細については、*DCC レジスタと DBGITR の外部ビューへのアクセス制御、v7 デバッグのみ* P. C10-21 を参照して下さい。
b. この列は、アクセスの制御に RXfull、RXfull_l、InstrCompl フラグのどれが使用されるかを示しています。アクセスは他のフラグの値には依存しません。
c. 書き込みがメモリマップされたインタフェース経由で行われ、ソフトウェアロックがセットされている場合、レジスタは読み出し専用で、アクセスに副作用はありません。これは次のことを意味します。
   DBGDTRRX、RXfull、RXfull_l、InstrCompl、InstrCompl_l は変更されません。
   アクセスは直ちに完了します。
   高速モードでは、命令は発行されません。
   詳細については、*メモリマップおよび外部デバッグインタフェースのアクセス許可の概要* P. C6-45 を参照して下さい。
d. RXfull が 1 の場合、DBGDTRRX、RXfull、RXfull_l は未知の値になります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

d. If RXfull is 1, the values of DBGDTRRX, RXfull, and RXfull_l become UNKNOWN.
e. If DBGDSCR.SDABORT_l, the Sticky Synchronous Data Abort bit, is set to 1, the instruction is not issued:
   InstrCompl and InstrCompl_l are unchanged
   the values of DBGDTRRX, RXfull and RXfull_l become UNKNOWN.
   For a description of the DBGDSCR.SDABORT_l bit, see *Debug Status and Control Register (DBGDSCR)* on page C10-10.
   Otherwise, the instruction is issued and InstrCompl and InstrCompl_l are cleared to 0.

C10.4.2 Target to Host Data Transfer Register (DBGDTRTX)

The Target to Host Data Transfer Register, DBGDTRTX, is used by the ARM processor to transfer data to an external host. For example it is used by a debug target to transfer data to the debugger.

The DBGDTRTX Register is:

- Debug register 35, at offset `0x08C`.
- A component of the *Debug Communication Channel* (DCC).
- Accessed through two views:
  — DBGDTRTXint, the internal view
  — DBGDTRTXext, the external view.
  See *Internal and external views of the DBGDSCR and the DCC registers* on page C6-21 for definitions of the internal and external views.
- When the Security Extensions are implemented, a Common register.

The behavior of accesses to the DBGDTRTX Register depends on:

- which view is being accessed
- the values of flags in the DCC.

For more information, see *Access to the DBGDTRTX Register* on page C10-44.

The architectural status of the DBGDTRRX Register depends on the Debug architecture version:

ARMv6 DBGDTRTX was previously named wDTR. DBGDTRTXext is not defined in ARMv6. However, the DBGDTRTXext functionality must be implemented as part of the external debug interface.

v7 Debug If implemented, the Extended CP14 interface instructions that access DBGDTRTXext are UNPREDICTABLE in Debug state. For more information, see *Internal and external views of the DBGDSCR and the DCC registers* on page C6-21 and *Extended CP14 interface* on page C6-33.

The format of the DBGDTRTX Register is:

31 … 0
Target to host data

Target to host data, bits [31:0]

One word of data for transfer from the debug target to the debug host.

The debug logic reset value of the DBGDTRTX Register is UNKNOWN.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

e. DBGDSCR.SDABORT_l（スティッキー同期データアボート ビット）が 1 にセットされている場合、命令は発行されません。
   InstrCompl と InstrCompl_l は変更されません。
   DBGDTRRX、RXfull、RXfull_l は未知の値になります。
   DBGDSCR.SDABORT_l ビットの説明については、*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。
   それ以外の場合、命令は発行され、InstrCompl と InstrCompl_l は 0 にクリアされます。

C10.4.2 ターゲットからホストへのデータ転送レジスタ (DBGDTRTX)

ターゲットからホストへのデータ転送レジスタ (DBGDTRTX) は、ARM プロセッサが外部ホストへデータを転送するために使用されます。例えば、デバッグターゲットがデバッガへデータを転送するために使用されます。

DBGDTRTX レジスタには、次の特徴があります。

- デバッグレジスタ 35、オフセット `0x08C`
- *デバッグ通信チャネル* (DCC) のコンポーネント
- 2 つのビューを経由してアクセスされます。
  – DBGDTRTXint、内部ビュー
  – DBGDTRTXext、外部ビュー
  内部ビューと外部ビューの定義については、*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6-21 を参照して下さい。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

DBGDTRTX レジスタへのアクセスの動作は、次の要因に依存します。

- アクセスされるビュー
- DCC のフラグの値

詳細については、*DBGDTRTX レジスタへのアクセス* P. C10-44 を参照して下さい。

DBGDTRRX レジスタのアーキテクチャ的なステータスは、デバッグアーキテクチャのバージョンによって異なります。

ARMv6 DBGDTRTX は、以前は wDTR という名前でした。ARMv6 では DBGDTRTXext は定義されていません。ただし、外部デバッグインタフェースの一部として DBGDTRTXext の機能を実装する必要があります。

v7 デバッグ DBGDTRTXext にアクセスする拡張 CP14 インタフェース命令は、それが実装されている場合、デバッグ状態では予測不能です。詳細については、*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6-21 と *拡張 CP14 インタフェース* P. C6-33 を参照して下さい。

DBGDTRTX レジスタのフォーマットは次のとおりです。

31 … 0
ターゲットからホストへのデータ

ターゲットからホストへのデータ、ビット [31:0]

デバッグターゲットからデバッグホストへ転送される 1 ワードのデータ。

デバッグロジック リセット時の DBGDTRTX レジスタの値は未知です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Access to the DBGDTRTX Register

The behavior on various accesses to the DBGDTRTX Register is described in the following tables:

- Table C10-7 shows the behavior of accesses to DBGDTRTXint
- Table C10-8 on page C10-45 shows the behavior of write accesses to DBGDTRTXext
- Table C10-9 on page C10-45 shows the behavior of read accesses to DBGDTRTXext.

To access the DBGDTRTXint Register you write the CP14 registers using either:

- an MCR instruction with <opc1> set to 0, <CRn> set to c0, <CRm> set to c5, and <opc2> set to 0
- an LDC instruction with <CRd> set to c5.

Both instructions write only one word to the DBGDTRTXint Register. For example:

```
MCR p14,0,<Rd>,c0,c5,0 ; Write DBGDTRTXint Register
LDC p14,c5,[<Rn>],#4   ; Read a word from memory and write it to the DBGDTRTXint Register
```

Table C10-7 Behavior of accesses to DBGDTRTXint

Access	TXfull	Action	New TXfull
Read	X	Not possible. There is no operation that reads from DBGDTRTXint.	-
Write	0	Writes value to DBGDTRTX.	1
	1	UNPREDICTABLE.	-

Note

- If the LDC instruction that writes to DBGDTRTXint aborts, the contents of DBGDTRTX and the value of the TXfull flag are UNKNOWN.
- The behavior on accesses to DBGDTRTXint does not depend on the value of TXfull_l
- Accesses to DBGDTRTXint do not update the value of TXfull_l.

Accesses to DBGDTRTXext can be made through:

- the Extended CP14 interface, if implemented
- the memory-mapped interface, if implemented
- the external debug interface.

Table C10-8 on page C10-45 shows the behavior of write accesses to DBGDTRTXext, and Table C10-9 on page C10-45 shows the behavior of read accesses to DBGDTRTXext.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

DBGDTRTX レジスタへのアクセス

DBGDTRTX レジスタに対する各種のアクセスの動作については、次の表を参照して下さい。

- DBGDTRTXint へのアクセスの動作を、表 C10-7 に示します。
- DBGDTRTXext への書き込みアクセスの動作を、P. C10-45 の表 C10-8 に示します。
- DBGDTRTXext への読み出しアクセスの動作を、P. C10-45 の表 C10-9 に示します。

DBGDTRTXint レジスタにアクセスするには、次のいずれかの命令を使用して CP14 レジスタに書き込みます。

- MCR 命令で <opc1> を 0、<CRn> を c0、<CRm> を c5、<opc2> を 0 に設定する。
- LDC 命令で <CRd> を c5 に設定する。

どちらの命令も、DBGDTRTXint レジスタに 1 ワードのみを書き込みます。次に例を示します。

```
MCR p14,0,<Rd>,c0,c5,0 ; Write DBGDTRTXint Register
LDC p14,c5,[<Rn>],#4   ; Read a word from memory and write it to the DBGDTRTXint Register
```

表 C10-7 DBGDTRTXint へのアクセスの動作

アクセス	TXfull	動作	新しい TXfull
読み出し	X	不可能。DBGDTRTXint を読み出す操作は存在しません。	–
書き込み	0	DBGDTRTX に値を書き込みます。	1
	1	予測不能	–

注

- DBGDTRTXint に書き込む LDC 命令がアボートした場合、DBGDTRTX の内容と TXfull フラグの値は未知です。
- DBGDTRTXint へのアクセス時の動作は、TXfull_l の値に依存しません。
- DBGDTRTXint へのアクセスでは、TXfull_l の値は更新されません。

DBGDTRTXext へのアクセスは、次のいずれかを経由して行うことができます。

- 拡張 CP14 インタフェース（実装されている場合）
- メモリマップされたインタフェース（実装されている場合）
- 外部デバッグインタフェース

DBGDTRTXext への書き込みアクセスの動作を P. C10-45 の表 C10-8 に、DBGDTRTXext への読み出しアクセスの動作を P. C10-45 の表 C10-9 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C10-8 Behavior of write accesses to DBGDTRTXext

Access mode[a]	Flag[b]	Flag value	Action	New TXfull and TXfull_l
X	X	X	Updates DBGDTRTX value[c]	Unchanged

a. For more information, see *Access controls on the external view of the DCC registers and DBGITR, v7 Debug only* on page C10-21.
b. This column indicates which of the TXfull, TXfull_l and InstrCompl flags are used to control the access. The access does not depend on the value of any other flags.
c. In the event of a race condition with writes to both DBGDTRTXint and DBGDTRTXext occurring, the result is UNPREDICTABLE. Writes to DBGDTRTXext must only be performed under controlled circumstances, for example when the processor is in Debug state.

Table C10-9 Behavior of read accesses to DBGDTRTXext

Access mode[a]	Flag[b]	Flag value	Action	New TXfull	New TXfull_l
Non-blocking	TXfull_l	0	Returns an UNKNOWN value.	Unchanged	Unchanged
		1	Returns DBGDTRTX contents	0[c]	0[c]
Stall	TXfull	0	Stalls[c] until (TXfull = 1)	-	-
		1	Returns DBGDTRTX contents	0[c]	0[c]
Fast	InstrCompl	0	Stalls[c] until (InstrCompl = 1)	-	-
		1	Returns DBGDTRTX contents[d] and issues the instruction in the DBGITR[c, e]	0[c, d, e]	0[c, d, e]

a. For more information, see *Access controls on the external view of the DCC registers and DBGITR, v7 Debug only* on page C10-21.
b. This column indicates which of the TXfull, TXfull_l and InstrCompl flags are used to control the access. The access does not depend on the value of any other flags.
c. If the read is made through the memory-mapped interface and the Software Lock is set, the registers are read-only and accesses have no side effects. This means that:
   TXfull, TXfull_l, InstrCompl, and InstrCompl_l remain unchanged
   the access completes immediately
   if TXfull==1, the access returns the contents of DBGDTRTX, otherwise it returns an UNKNOWN value
   in Fast mode no instruction is issued.
   For more information, see *Permission summaries for memory-mapped and external debug interfaces* on page C6-45.
d. If TXfull is 0, this returns an UNKNOWN value and the values of DBGDTRTX, TXfull and TXfull_l become UNKNOWN.
e. The value returned is the value of DBGDTRTX before the instruction issued modifies the state of the processor.
   If DBGDSCR.SDABORT_l, the Sticky Synchronous Data Abort bit, is set to 1, the instruction is not issued, InstrCompl and InstrCompl_l remain unchanged, and the values of TXfull and TXfull_l become UNKNOWN. For a description of the DBGDSCR.SDABORT_l bit, see *Debug Status and Control Register (DBGDSCR)* on page C10-10.
   Otherwise, the instruction is issued and InstrCompl and InstrCompl_l are cleared to 0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C10-8 DBGDTRTXext への書き込みアクセスの動作

アクセスモード[a]	フラグ[b]	フラグの値	動作	新しい TXfull と TXfull_l
X	X	X	DBGDTRTX の値を更新します。[c]	変更なし

a. 詳細については、*DCC レジスタと DBGITR の外部ビューへのアクセス制御、v7 デバッグのみ* P. C10-21 を参照して下さい。
b. この列は、アクセスの制御に TXfull、TXfull_l、InstrCompl フラグのどれが使用されるかを示しています。アクセスは他のフラグの値には依存しません。
c. DBGDTRTXint と DBGDTRTXext 両方への書き込みが競合する状況が発生した場合、結果は予測不能です。DBGDTRTXext への書き込みは、プロセッサがデバッグ状態のときなど、制御された状況でのみ実行する必要があります。

表 C10-9 DBGDTRTXext への読み出しアクセスの動作

アクセスモード[a]	フラグ[b]	フラグの値	動作	新しい TXfull	新しい TXfull_l
非ブロッキング	TXfull_l	0	未知 (UNKNOWN) の値が返されます。	変更なし	変更なし
		1	DBGDTRTX の内容が返されます。	0[c]	0[c]
ストール	TXfull	0	(TXfull = 1) になるまでストールします。[c]	–	–
		1	DBGDTRTX の内容が返されます。	0[c]	0[c]
高速	InstrCompl	0	(InstrCompl = 1) になるまでストールします。[c]	–	–
		1	DBGDTRTX の内容を返し[d] DBGITR の命令を発行します。[c, e]	0[c, d, e]	0[c, d, e]

a. 詳細については、*DCC レジスタと DBGITR の外部ビューへのアクセス制御、v7 デバッグのみ* P. C10-21 を参照して下さい。
b. この列は、アクセスの制御に TXfull、TXfull_l、InstrCompl フラグのどれが使用されるかを示しています。アクセスは他のフラグの値には依存しません。
c. 読み出しがメモリマップされたインタフェース経由で行われ、ソフトウェアロックが設定されている場合、レジスタは読み出し専用で、アクセスに副作用はありません。これは次のことを意味します。
   TXfull、TXfull_l、InstrCompl、InstrCompl_l は変更されません。
   アクセスは直ちに完了します。
   TXfull = 1 の場合、アクセスにより DBGDTRTX の内容が返されます。それ以外の場合、未知の値が返されます。
   高速モードでは、命令は発行されません。
   詳細については、*メモリマップおよび外部デバッグインタフェースのアクセス許可の概要* P. C6-45 を参照して下さい。
d. TXfull が 0 の場合、返される値は未知で、DBGDTRTX、TXfull、TXfull_l は未知の値になります。
e. 返される値は、発行された命令によりプロセッサの状態が変更される前の DBGDTRTX の値です。
   DBGDSCR.SDABORT_l（スティッキー同期データアボート ビット）が 1 にセットされている場合、この命令は発行されず、InstrCompl と InstrCompl_l は変更されず、TXfull と TXfull_l は未知の値になります。
   DBGDSCR.SDABORT_l ビットの説明については、*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。
   それ以外の場合、命令は発行され、InstrCompl と InstrCompl_l は 0 にクリアされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.4.3 Instruction Transfer Register (DBGITR)

The Instruction Transfer Register, DBGITR, enables external debugger to transfer ARM instructions to the processor for execution when the processor is in Debug state.

The DBGITR is:

- Debug register 33, at offset `0x084`.
- A write-only register. However, accesses to the DBGITR also depend on:
  - — the processor state
  - — the values of the DBGDSCR.ExtDCCmode and DBGDSCR.ITRen fields, see *Debug Status and Control Register (DBGDSCR)* on page C10-10
  - — the values of the DCC and InstrCompl_l flags.

  For more information, see *Accesses to the DBGITR*.
- When the Security Extensions are implemented, a Common register.

The architectural status of the DBGITR depends on the Debug architecture version:

ARMv6 DBGITR is not defined in ARMv6. However, it might form part of the external debug interface.

v7 Debug Writes through the Extended CP14 interface of the CP14 register that maps to the DBGITR are UNDEFINED in User mode and UNPREDICTABLE in privileged modes.

The format of the DBGITR is:

31 … 0
ARM instruction to execute on the processor

ARM instruction to execute on the processor, bits [31:0]

The 32-bit encoding of an ARM instruction to execute on the processor.

The debug logic reset value of the DBGITR is UNKNOWN.

Accesses to the DBGITR

Writes to the DBGITR are UNPREDICTABLE when:

- the processor is in Non-debug state
- DBGDSCR.ITRen is set to 0.

Table C10-10 on page C10-47 shows the behavior of writes to the DBGITR when in Debug state with the DBGDSCR.ITRen flag is set to 1.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.4.3 命令転送レジスタ (DBGITR)

命令転送レジスタ (DBGITR) は、プロセッサがデバッグ状態のとき、外部デバッガが ARM 命令を実行のためプロセッサへ転送するために使用されます。

DBGITR の特徴は次のとおりです。

- デバッグレジスタ 33、オフセット `0x084`
- 書き込み専用レジスタ。ただし、DBGITR へのアクセスは次の要因にも依存します。
  - – プロセッサの状態
  - – DBGDSCR.ExtDCCmode および DBGDSCR.ITRen フィールドの値。*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。
  - – DCC および InstrCompl_l フラグの値

  詳細については、*DBGITR へのアクセス*を参照して下さい。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

DBGITR のアーキテクチャ的なステータスは、デバッグアーキテクチャのバージョンによって異なります。

ARMv6 ARMv6 では DBGITR は定義されていません。ただし、外部デバッグインタフェースの一部を構成している場合があります。

v7 デバッグ CP14 レジスタの拡張 CP14 インタフェースを経由し、DBGITR にマッピングされる書き込みは、ユーザモードでは未定義で、特権モードでは予測不能です。

DBGITR のフォーマットは次のとおりです。

31 … 0
プロセッサで実行するための ARM 命令

プロセッサで実行するための ARM 命令、ビット [31:0]

プロセッサで実行するための ARM 命令の 32 ビットエンコード。

デバッグロジック リセット時の DBGITR の値は未知です。

DBGITR へのアクセス

次の場合、DBGITR への書き込みは予測不能です。

- プロセッサが非デバッグ状態である。
- DBGDSCR.ITRen が 0 にセットされている。

デバッグ状態で、DBGDSCR.ITRen フラグが 1 にセットされているときの、DBGITR への書き込みの動作を、P. C10-47 の表 C10-10 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C10-10 Behavior of write accesses to DBGITR

Access mode[a]	Flag[b]	Flag value	Action	New InstrCompl	New InstrCompl_l
Non-blocking	InstrCompl_l	0	Write is ignored	Unchanged	Unchanged
		1	Issue instruction[c]	0[c]	0[c]
Stall	InstrCompl	0	Stall until (InstrCompl = 0)	-	-
		1	Issue instruction[c]	0[c]	0[c]
Fast	Not applicable	-	Save instruction in DBGITR[d]	-	-

a. For more information, see *Access controls on the external view of the DCC registers and DBGITR, v7 Debug only* on page C10-21.
b. This column indicates which flag controls the access. The access does not depend on the value of any other flag.
c. If DBGDSCR.SDABORT_l, the Sticky Synchronous Data Abort bit, is set to 1, the instruction is not issued and InstrCompl remains unchanged. For a description of the DBGDSCR.SDABORT_l bit, see *Debug Status and Control Register (DBGDSCR)* on page C10-10.
d. The instruction is saved in the DBGITR and is issued on a read of DBGDTRTXext or a write of DBGDTRRXext. For more information, see *Access controls on the external view of the DCC registers and DBGITR, v7 Debug only* on page C10-21.

If the write is made through the memory-mapped interface and the Software Lock is set to 1, writes to the DBGITR are ignored and have no other side-effects. This means that:

- the DBGITR, and the InstrCompl and InstrCompl_l flags, remain unchanged
- no instruction is issued.

For more information, see *Permission summaries for memory-mapped and external debug interfaces* on page C6-45.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C10-10 DBGITR への書き込みアクセスの動作

アクセスモード[a]	フラグ[b]	フラグの値	動作	新しい InstrCompl	新しい InstrCompl_l
非ブロッキング	InstrCompl_l	0	書き込みは無視されます。	変更なし	変更なし
		1	命令を発行します。[c]	0[c]	0[c]
ストール	InstrCompl	0	(InstrCompl = 0) になるまでストールします。	–	–
		1	命令を発行します。[c]	0[c]	0[c]
高速	該当せず	–	DBGITR に命令を保存します。[d]	–	–

a. 詳細については、*DCC レジスタと DBGITR の外部ビューへのアクセス制御、v7 デバッグのみ* P. C10-21 を参照して下さい。
b. この列は、どのフラグがアクセスを制御するかを示しています。アクセスは他のフラグの値には依存しません。
c. DBGDSCR.SDABORT_l（スティッキー同期データアボート ビット）が 1 にセットされている場合、命令は発行されず、InstrCompl は変更されません。DBGDSCR.SDABORT_l ビットの説明については、*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。
d. 命令は DBGITR に保存され、DBGDTRTXext の読み出し、または DBGDTRRXext への書き込みが行われたときに発行されます。詳細については、*DCC レジスタと DBGITR の外部ビューへのアクセス制御、v7 デバッグのみ* P. C10-21 を参照して下さい。

書き込みがメモリマップされたインタフェース経由で行われ、ソフトウェアロックが 1 にセットされている場合、DBGITR への書き込みは無視され、他の副作用はありません。これは次のことを意味します。

- DBGITR と、InstrCompl および InstrCompl_l フラグは変更されません。
- 命令は発行されません。

詳細については、*メモリマップおよび外部デバッグインタフェースのアクセス許可の概要* P. C6-45 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.5 Software debug event registers

This section contains the following subsections:

- *Breakpoint Value Registers (DBGBVR)*
- *Breakpoint Control Registers (DBGBCR)* on page C10-49
- *Watchpoint Value Registers (DBGWVR)* on page C10-60
- *Watchpoint Control Registers (DBGWCR)* on page C10-61
- *Vector Catch Register (DBGVCR)* on page C10-67.

In addition, when the OS Save and Restore mechanism is implemented, the Event Catch Register can be used to enable generation of a debug event when the OS Lock is unlocked, see *Event Catch Register (DBGECR)* on page C10-78.

C10.5.1 Breakpoint Value Registers (DBGBVR)

A Breakpoint Value Register, DBGBVR, holds a value for use in breakpoint matching. The value is either an *Instruction Virtual Address* (IVA) or a Context ID. Each DBGBVR is associated with a DBGBCR to form a *Breakpoint Register Pair* (BRP). DBGBVRn is associated with DBGBCRn to form BRPn, where n takes the values from 0 to 15. A debug event is generated when an instruction that matches the BRP is committed for execution. For more information, see *Breakpoint debug events* on page C3-5.

A breakpoint can be set on any one of:

- an IVA match or mismatch
- a Context ID match
- an IVA match or mismatch occurring with a Context ID match.

For the third case:

- two BRPs must be linked, see *Breakpoint Control Registers (DBGBCR)* on page C10-49.
- a debug event is generated when, on the same instruction, both:
  - — the IVA matches or mismatches, as required
  - — the Context ID matches.

See *Memory addresses* on page C3-23 for a definition of the IVA used to program a DBGBVR.

Note

Some BRPs might not support Context ID comparison. For more information, see the description of the DBGDIDR.CTX_CMPs field in *Debug ID Register (DBGDIDR)* on page C10-3.

The DBGBVRs are:

- debug registers 64-79, at offsets `0x100-0x13C`
- read/write registers
- when the Security Extensions are implemented, Common registers.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C10.5 ソフトウェアデバッグイベント レジスタ

このセクションは、次のサブセクションから構成されています。

- *ブレークポイント値レジスタ (DBGBVR)*
- *ブレークポイント制御レジスタ (DBGBCR)* P. C10- 49
- *ウォッチポイント値レジスタ (DBGWVR)* P. C10- 60
- *ウォッチポイント制御レジスタ (DBGWCR)* P. C10- 61
- *ベクタキャッチ レジスタ (DBGVCR)* P. C10- 67

これらに加えて、OS 保存および復元機構が実装されている場合、イベントキャッチ レジスタを使用して、OS ロックが解除されたときにデバッグイベントを生成することができます。*イベントキャッチ レジスタ (DBGECR)* P. C10- 78 を参照して下さい。

C10.5.1 ブレークポイント値レジスタ (DBGBVR)

ブレークポイント値レジスタ (DBGBVR) は、ブレークポイント一致に使用される値を保持します。値は*命令仮想アドレス*(IVA) またはコンテキスト ID です。それぞれの DBGBVR は DBGBCR と関連付けられ、*ブレークポイントレジスタ ペア*(BRP) を構成します。DBGBVRn は DBGBCRn と関連付けられ、BRPn を構成します。ここで、n の値は 0 ～ 15 です。BRP に一致する命令が実行にコミットされると、デバッグイベントが生成されます。詳細については、*ブレークポイントデバッグ イベント* P. C3- 5 を参照して下さい。

ブレークポイントは、次のいずれかに設定できます。

- IVA 一致または不一致
- コンテキスト ID 一致
- コンテキスト ID 一致とともに発生する IVA 一致または不一致

3 番目の場合、次の規則が適用されます。

- 2 つの BRP はリンクしている必要があります。*ブレークポイント制御レジスタ (DBGBCR)* P. C10- 49 を参照して下さい。
- デバッグイベントは、同じ命令で次の条件の両方が満たされたときに生成されます。
  - – 要求に応じて、IVA 一致または不一致
  - – コンテキスト ID 一致

DBGBVR をプログラムするために使用される IVA の定義については、*メモリアドレス* P. C3- 23 を参照して下さい。

注

一部の BRP では、コンテキスト ID の比較がサポートされていません。詳細については、*デバッグ ID レジスタ (DBGDIDR)* P. C10- 3 の DBGDIDR.CTX_CMP フィールドについての説明を参照して下さい。

DBGBVR の特徴は次のとおりです。

- デバッグレジスタ 64 ～ 79、オフセット `0x100` ～ `0x13C`
- 読み出し / 書き込みレジスタ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When used for IVA comparison the format of a DBGBVR is:

31 … 2	1	0
Breakpoint address[31:2]	0	0

When used for Context ID comparison the format of a DBGBVR is:

31 … 0
Context ID[31:0]

Bits [31:2] Bits [31:2] of the value for comparison. Either IVA[31:2] or ContextID[31:2].

Bits [1:0], when register used for IVA comparison

Must be written as 0b00, otherwise the generation of Breakpoint debug events is UNPREDICTABLE

Bits [1:0], when register used for Context ID comparison

Bits [1:0] of the value for comparison, ContextID[1:0].

If the BRP does not support Context ID comparison then bits [1:0] are UNK/SBZP.

The debug logic reset values of all bits of a DBGBVR are UNKNOWN.

C10.5.2 Breakpoint Control Registers (DBGBCR)

A Breakpoint Control Register, DBGBCR, holds control information for a breakpoint. Each DBGBCR is associated with a DBGBVR to form a *Breakpoint Register Pair* (BRP). For more information about BRPs and the possible breakpoints, see *Breakpoint Value Registers (DBGBVR)* on page C10-48.

The DBGBCRs are:

- debug registers 80-95, at offsets `0x140-0x17C`
- read/write registers
- when the Security Extensions are implemented, Common registers.

The format of a DBGBCR, in v7 Debug, is:

See the bit descriptions for the differences in other Debug architecture versions.

Bits [31:29,23,13:9,4:3]

Reserved, UNK/SBZP.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

IVA 比較に使用される場合、DBGBVR のフォーマットは次のとおりです。

31 … 2	1	0
ブレークポイントアドレス [31:2]	0	0

コンテキスト ID 比較に使用される場合、DBGBVR のフォーマットは次のとおりです。

31 … 0
コンテキスト ID[31:0]

ビット [31:2] 比較する値のビット [31:2]。IVA[31:2] または ContextID[31:2] です。

ビット [1:0]、レジスタが IVA 比較に使用される場合

0b00 として書き込まれる必要があります。そうでない場合、ブレークポイント デバッグ イベントの生成は予測不能です。

ビット [1:0]、レジスタがコンテキスト ID 比較に使用される場合

比較する値のビット [1:0]、ContextID[1:0]。

BRP がコンテキスト ID 比較をサポートしていない場合、ビット [1:0] は UNK/SBZP です。

DBGBVR のすべてのビットについて、デバッグロジック リセット時の値は未知です。

C10.5.2 ブレークポイント制御レジスタ (DBGBCR)

ブレークポイント制御レジスタ (DBGBCR) は、ブレークポイントの制御情報を保持します。それぞれの DBGBCR は DBGBVR と関連付けられ、*ブレークポイントレジスタ ペア*(BRP) を構成します。BRP と可能なブレークポイントの詳細については、*ブレークポイント値レジスタ (DBGBVR)* P. C10- 48 を参照して下さい。

DBGBCR の特徴は次のとおりです。

- デバッグレジスタ 80 ～ 95、オフセット `0x140` ～ `0x17C`
- 読み出し / 書き込みレジスタ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

v7 デバッグでの DBGBCR のフォーマットは次のとおりです。

デバッグアーキテクチャの他のバージョンにおける相違点については、ビットの説明を参照して下さい。

ビット [31:29、23、13:9、4:3]

予約、UNK/SBZP

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Address range mask, bits [28:24], v7 Debug

In v7 Debug, whether address range masking is supported is IMPLEMENTATION DEFINED. If it is not supported these bits are RAZ/WI.

If address range masking is supported, this field can be used to break on a range of addresses by masking lower order address bits out of the breakpoint comparison. The value of this field is the number of low order bits of the address that are masked off, except that values of 1 and 2 are reserved. Therefore, the meaning of Breakpoint address range mask values are:

0b00000 No mask

0b00001 Reserved

0b00010 Reserved

0b00011 `0x00000007` mask for instruction address, three bits masked

0b00100 `0x0000000F` mask for instruction address, four bits masked

0b00101 `0x0000001F` mask for instruction address, five bits masked

. .

. .

. .

0b11111 `0x7FFFFFFF` mask for instruction address, 31 bits masked.

This field must be programmed to 0b00000 if either:

- this BRP is programmed for Context ID comparison
- the Byte address select field is programmed to a value other than 0b1111.

If this is not done, the generation of Breakpoint debug events is UNPREDICTABLE.

If this field is not zero, the DBGBVR bits that are not included in the comparison must be zero, otherwise the generation of Breakpoint debug events is UNPREDICTABLE.

For more information about the use of this field see *Breakpoint address range masking behavior, v7 Debug* on page C3-9.

Bits [28:24], v6 Debug and v6.1 Debug

Reserved, UNK/SBZP.

DBGBVR meaning, bits [22:20]

This field controls the behavior of Breakpoint debug event generation. This includes the meaning of the value held in the associated DBGBVR, whether it is an IVA or a Context ID. Each bit of this field has particular significance, and there can be restrictions on the values of bits [22:21]:

Bit [22], Match or mismatch

This bit is set to 1 for a mismatch comparison.

This bit is not supported and is UNK/SBZP in v6 Debug.

For more information about IVA mismatching see *Additional considerations for IVA mismatch breakpoints* on page C3-13.

The Debug architecture does not support Context ID mismatch comparisons.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アドレス範囲マスク、ビット [28:24]、v7 デバッグ

v7 デバッグでは、アドレス範囲マスクがサポートされるかどうかは実装定義です。サポートされない場合、これらのビットは RAZ/WI です。

アドレス範囲マスクがサポートされている場合、このフィールドを使用して、ブレークポイント比較アドレスの下位ビットをマスクにより除外し、一定範囲のアドレスにブレークポイントを設定できます。このフィールドの値は、アドレスのうちマスクにより除外される下位ビットの数を示します。ただし、値 1 と 2 は予約されています。したがって、ブレークポイントアドレス範囲マスクの値の意味は次のとおりです。

0b00000 マスクなし

0b00001 予約

0b00010 予約

0b00011 命令アドレスに `0x00000007` マスクが適用され、3 ビットがマスクされます。

0b00100 命令アドレスに `0x0000000F` マスクが適用され、4 ビットがマスクされます。

0b00101 命令アドレスに `0x0000001F` マスクが適用され、5 ビットがマスクされます。

. .

. .

. .

0b11111 命令アドレスに `0x7FFFFFFF` マスクが適用され、31 ビットがマスクされます。

次のいずれかの条件が当てはまる場合、このフィールドを 0b00000 にプログラムする必要があります。

- この BRP をコンテキスト ID 比較用にプログラムする場合
- バイトアドレス選択フィールドが、0b1111 以外の値にプログラムされている場合

この操作を行わない場合、ブレークポイントデバッグ イベントの生成は予測不能です。

このフィールドが 0 以外のとき、比較に含まれない DBGBVR ビットは 0 の必要があります。そうでない場合、ブレークポイントデバッグ イベントの生成は予測不能です。

このフィールドの使用方法の詳細については、*v7 デバッグでのブレークポイントアドレス範囲マスクの動作* P. C3-9 を参照して下さい。

ビット [28:24]、v6 デバッグおよび v6.1 デバッグのみ

予約、UNK/SBZP

DBGBVR の意味、ビット [22:20]

このフィールドは、ブレークポイントデバッグ イベント生成の動作を制御します。これには、関連付けられている DBGBVR に保持されている値の意味、それが IVA とコンテキスト ID のどちらであるかが含まれます。このフィールドの各ビットには特定の意味があり、ビット [22:21] の値には制限が存在する可能性があります。

ビット [22]、一致または不一致

このビットは、不一致比較のときに 1 にセットされます。

このビットは v6 デバッグではサポートされておらず、UNK/SBZP です。

IVA 不一致の詳細については、*IVA 不一致ブレークポイントについての追加の考慮事項* P. C3-13 を参照して下さい。

デバッグアーキテクチャは、コンテキスト ID の不一致比較をサポートしていません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Bit [21], IVA or Context ID comparison

This bit is set to 1 for a Context ID comparison.

This bit is UNK/SBZP for BRPs that do not support Context ID comparison. In this case field values of 0b010 and 0b011 are not supported.

Bit [20], Unlinked or Linked comparison

This bit is set to 1 if this BRP is linked to another BRP to set a breakpoint that requires both IVA and Context ID comparison.

For more information about IVA matching and mismatching see:

- *Byte address selection behavior on IVA match or mismatch* on page C10-55
- *Breakpoint address range masking behavior, v7 Debug* on page C3-9
- *IVA comparisons and instruction length* on page C3-10.

The possible values of the *DBGBVR meaning* field are:

0b000 Unlinked Instruction Virtual Address match

Compare:

- the byte address select bits, bits [8:5], and the associated DBGBVR[31:2], against the IVA of the instruction
- the security state control and privileged mode control bits, bits [15:14,2:1], against the state of the processor.

Generate a Breakpoint debug event on a joint IVA match and state match.

DBGBCR[19:16] must be programmed to 0b0000, otherwise the generation of Breakpoint debug events is UNPREDICTABLE.

0b001 Linked Instruction Virtual Address match

Compare:

- the byte address select bits, bits [8:5], and the associated DBGBVR[31:2], against the IVA of the instruction
- the security state control and privileged mode control bits, bits [15:14,2:1], against the state of the processor.

This BRP is linked with the BRP indicated by DBGBCR[19:16]. Generate a Breakpoint debug event on a joint IVA match, Context ID match and state match. For more information, see *Linked comparisons* on page C10-59.

0b010 Unlinked Context ID match

Compare:

- the associated DBGBVR[31:0] against the Context ID in the CONTEXTIDR
- the security state control and privileged mode control bits, bits [15:14,2:1], against the state of the processor.

This BRP is not linked with any other one. Generate a Breakpoint debug event on a joint Context ID match and state match.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ビット [21]、IVA またはコンテキスト ID 比較

このビットは、コンテキスト ID 比較のときに 1 にセットされます。

コンテキスト ID 比較をサポートしない BRP では、このビットは UNK/SBZP です。この場合、フィールドの値 0b010 および 0b011 はサポートされていません。

ビット [20]、リンクなしまたはリンク付き比較

このビットは、この BRP が別の BRP にリンクされ、IVA とコンテキスト ID 比較の両方を必要とするブレークポイントの設定に使用される場合に 1 にセットされます。

IVA 一致と不一致の詳細については、次のセクションを参照して下さい。

- *IVA 一致または不一致時のバイトアドレス選択の動作* P. C10-55
- *v7 デバッグでのブレークポイントアドレス範囲マスクの動作* P. C3-9
- *IVA 比較と命令長* P. C3-10

*DBGBVR の意味*フィールドの可能な値は次のとおりです。

0b000 リンクなし命令仮想アドレス一致

次の比較を行います。

- バイトアドレス選択ビット（ビット [8:5]）と、関連付けられている DBGBVR[31:2] を、命令の IVA と比較します。
- セキュリティ状態制御および特権モード制御ビット（ビット [15:14、2:1]）を、プロセッサの状態と比較します。

IVA 一致と状態一致の組み合わせにより、ブレークポイントデバッグ イベントを生成します。

DBGBCR[19:16] は 0b0000 にプログラムする必要があります。そうでない場合、ブレークポイントデバッグ イベントの生成は予測不能です。

0b001 リンク付き命令仮想アドレス一致

次の比較を行います。

- バイトアドレス選択ビット（ビット [8:5]）と、関連付けられている DBGBVR[31:2] を、命令の IVA と比較します。
- セキュリティ状態制御および特権モード制御ビット（ビット [15:14、2:1]）を、プロセッサの状態と比較します。

この BRP は、DBGBCR[19:16] で示される BRP とリンクされています。IVA 一致、コンテキスト ID 一致、状態一致の組み合わせにより、ブレークポイントデバッグ イベントを生成します。詳細については、*リンク付き比較* P. C10-59 を参照して下さい。

0b010 リンクなしのコンテキスト ID 一致

次の比較を行います。

- 関連付けられている DBGBVR[31:0] を、CONTEXTIDR のコンテキスト ID と比較します。
- セキュリティ状態制御および特権モード制御ビット（ビット [15:14、2:1]）を、プロセッサの状態と比較します。

この BRP は他の BRP とリンクされていません。コンテキスト ID 一致と状態一致の組み合わせにより、ブレークポイントデバッグ イベントを生成します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DBGBCR[8:5] must be programmed to 0b1111 and DBGBCR[19:16] must be programmed to 0b0000, otherwise the generation of Breakpoint debug events is UNPREDICTABLE.

Note

See *Unpredictable behavior on Software debug events* on page C3-24 for additional restrictions for this type of breakpoint when using Monitor debug-mode.

0b011 Linked Context ID match

Compare the associated DBGBVR[31:0] against the Context ID in the CONTEXTIDR.

At least one other BRP or WRP is linked with this BRP. Generate a Breakpoint or Watchpoint debug event jointly on:

- the IVA match or mismatch or DVA match, defined by the linked BRP or WRP
- the Context ID match defined by this BRP.

If no BRP or WRP of the correct type is linked to this BRP, no Breakpoint or Watchpoint debug events are generated for this BRP.

For more information about the programming required for a Linked Context ID match see *Linked comparisons* on page C10-59.

0b100 Unlinked Instruction Virtual Address mismatch

Compare:

- the byte address select bits, bits [8:5], and the associated DBGBVR[31:2], against the IVA of the instruction
- the security state control and privileged mode control bits, bits [15:14,2:1], against the state of the processor.

Generate a Breakpoint debug event on a joint IVA mismatch (IVA not equal) and state match.

DBGBCR[19:16] must be programmed to 0b0000, otherwise the generation of Breakpoint debug events is UNPREDICTABLE.

Note

- Unlinked IVA mismatch is not supported in v6 Debug.
- See *Unpredictable behavior on Software debug events* on page C3-24 for additional restrictions for this type of breakpoint when using Monitor debug-mode.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

DBGBCR[8:5] は 0b1111 に、DBGBCR[19:16] は 0b0000 にプログラムする必要があります。そうでない場合、ブレークポイントデバッグ イベントの生成は予測不能です。

注

モニタデバッグ モードの使用時に、このタイプのブレークポイントに課せられる追加の制限については、*ソフトウェアデバッグ イベントでの予測不能な動作* P. C3-24 を参照して下さい。

0b011 リンク付きのコンテキスト ID 一致

関連付けられている DBGBVR[31:0] を、CONTEXTIDR のコンテキスト ID と比較します。

この BRP には、最低 1 つの他の BRP または WRP がリンクされています。次の条件の組み合わせにより、ブレークポイントまたはウォッチポイントデバッグ イベントを生成します。

- リンク付き BRP または WRP により定義される、IVA 一致または不一致、あるいは DVA 一致
- この BRP により定義されるコンテキスト ID 一致

この BRP に正しいタイプの BRP または WRP がリンクされていない場合、この BRP についてブレークポイントまたはウォッチポイントデバッグ イベントは生成されません。

リンク付きコンテキスト ID 一致に必要なプログラムの詳細については、*リンク付き比較* P. C10-59 を参照して下さい。

0b100 リンクなし命令仮想アドレス不一致

次の比較を行います。

- バイトアドレス選択ビット（ビット [8:5]）と、関連付けられている DBGBVR[31:2] を、命令の IVA と比較します。
- セキュリティ状態制御および特権モード制御ビット（ビット [15:14、2:1]）を、プロセッサの状態と比較します。

IVA 不一致（IVA が等しくない）と状態一致の組み合わせにより、ブレークポイントデバッグ イベントを生成します。

DBGBCR[19:16] は 0b0000 にプログラムする必要があります。そうでない場合、ブレークポイントデバッグ イベントの生成は予測不能です。

注

- v6 デバッグでは、リンクなし IVA 不一致はサポートされていません。
- モニタデバッグ モードの使用時に、このタイプのブレークポイントに課せられる追加の制限については、*ソフトウェアデバッグ イベントでの予測不能な動作* P. C3-24 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b101 Linked Instruction Virtual Address mismatch

Compare:

- the byte address select bits, bits [8:5], and the associated DBGBVR[31:2], against the IVA of the instruction
- the security state control and privileged mode control bits, bits [15:14,2:1], against the state of the processor.

This BRP is linked with the BRP indicated by DBGBCR[19:16]. Generate a Breakpoint debug event on a joint IVA mismatch (IVA not equal), state match and Context ID match. For more information, see *Linked comparisons* on page C10-59.

Note

- Linked IVA mismatch is not supported in v6 Debug.
- See *Unpredictable behavior on Software debug events* on page C3-24 for additional restrictions for this type of breakpoint when using Monitor debug-mode.

0b11x Reserved

Generation of Breakpoint debug events is UNPREDICTABLE.

Summary of breakpoint generation options on page C10-58 shows what comparisons are made for each permitted value of this field.

Linked BRP number, bits [19:16]

If this BRP is programmed for Linked IVA match or mismatch then this field must be programmed with the number of the BRP that holds the Context ID to be used for the combined IVA and Context ID comparison, otherwise, this field must be programmed to 0b0000.

If this field is programmed with a value other than zero or the number of a BRP that supports Context ID comparison then reading this register returns an UNKNOWN value for this field.

The generation of Breakpoint debug events is UNPREDICTABLE if either:

- this BRP is not programmed for Linked IVA match or mismatch and this field is not programmed to 0b0000
- this BRP is programmed for Linked IVA match or mismatch and the BRP indicated by this field does not support Context ID comparison or is not programmed for Linked Context ID match.

See also *Generation of debug events* on page C3-40.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b101 リンク付き命令仮想アドレス不一致

次の比較を行います。

- バイトアドレス選択ビット（ビット [8:5]）と、関連付けられている DBGBVR[31:2] を、命令の IVA と比較します。
- セキュリティ状態制御および特権モード制御ビット（ビット [15:14、2:1]）を、プロセッサの状態と比較します。

この BRP は、DBGBCR[19:16] で示される BRP とリンクされています。IVA 不一致（IVA が等しくない）、状態一致、コンテキスト ID 一致の組み合わせにより、ブレークポイントデバッグ イベントを生成します。詳細については、*リンク付き比較* P. C10-59 を参照して下さい。

注

- v6 デバッグでは、リンク付き IVA 不一致はサポートされていません。
- モニタデバッグ モードの使用時に、このタイプのブレークポイントに課せられる追加の制限については、*ソフトウェアデバッグ イベントでの予測不能な動作* P. C3-24 を参照して下さい。

0b11x 予約

ブレークポイントデバッグ イベントの生成は予測不能です。

このフィールドに許可されるそれぞれの値について、どのような比較が行われるかを、*ブレークポイント生成オプションの要約* P. C10-58 に示します。

リンク付き BRP 番号、ビット [19:16]

この BRP がリンク付き IVA 一致または不一致用にプログラムされる場合、このフィールドには、IVA とコンテキスト ID の複合比較に使用されるコンテキスト ID を保持する、BRP の番号をプログラムする必要があります。それ以外の場合、このフィールドを 0b0000 にプログラムする必要があります。

このフィールドが、0 またはコンテキスト ID 比較をサポートする BRP 番号のいずれでもない値にプログラムされている場合、このレジスタを読み出したときにこのフィールドについて返される値は未知です。

次のいずれかの場合、ブレークポイントデバッグ イベントの生成は予測不能です。

- この BRP が、リンク付き IVA 一致または不一致用にプログラムされておらず、このフィールドが 0b0000 にプログラムされていない。
- この BRP が、リンク付き IVA 一致または不一致用にプログラムされており、このフィールドによって示される BRP がコンテキスト ID 比較をサポートしていない、またはリンク付きコンテキスト ID 一致用にプログラムされていない。

デバッグ イベントの生成 P. C3-40 も参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Security state control, bits [15:14], when the Security Extensions are implemented

Note

The Security Extensions cannot be implemented with v6 Debug.

When a processor implements the Security Extensions, these bits enable the breakpoint to be conditional on the security state of the processor:

0b00 breakpoint generated on match in both Non-secure state and Secure state

0b01 breakpoint generated on match only in Non-secure state

0b10 breakpoint generated on match only in Secure state

0b11 Reserved.

This field must be programmed to 0b00 if the DBGBVR meaning field, bits [22:20], is programmed for Linked Context ID match.

For more information about breakpoint matching when this field is set to a value other than 0b00, see *About security state control* on page C10-66.

See also *Generation of debug events* on page C3-40.

Bits [15:14], when the Security Extensions are not implemented

Reserved, UNK/SBZP.

Byte address select, bits [8:5]

This field enables match or mismatch comparisons on only certain bytes of the word address held in the DBGBVR. The operation of this field depends also on:

- the *DBGBVR meaning* field being programmed for IVA match or mismatch
- in v7 Debug, the Address range mask field being programmed to 0b00000, no mask
- the instruction set state of the processor, indicated by the CPSR.J and CPSR.T bits.

For details of the use of this field see *Byte address selection behavior on IVA match or mismatch* on page C10-55.

This field must be programmed to 0b1111 if either:

- the DBGBVR meaning field, bits [22:20], is programmed for Linked or Unlinked Context ID match
- in v7 Debug, the Address range mask field, bits [28:24], is programmed to a value other than 0b00000.

If this is not done, the generation of Breakpoint debug events is UNPREDICTABLE.

Privileged mode control, bits [2:1]

This field enables breakpoint matching conditional on the mode of the processor. Possible values of this field are:

0b00 Match any of User, System and Supervisor modes.
This value is supported in v7 Debug only.

0b01 Match in any privileged mode.

0b10 Match in User mode only.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

セキュリティ状態制御、ビット [15:14]、セキュリティ拡張機能が実装されている場合

注

v6 デバッグでは、セキュリティ拡張機能を実装できません。

プロセッサにセキュリティ拡張機能が実装されている場合、これらのビットにより、プロセッサのセキュリティ状態についてブレークポイントを条件付きにできます。

0b00 非セキュア状態とセキュア状態の両方で、一致によりブレークポイントが生成されます。

0b01 非セキュア状態での一致でのみブレークポイントが生成されます。

0b10 セキュア状態での一致でのみブレークポイントが生成されます。

0b11 予約

DBGBVR の意味フィールド（ビット [22:20]）がリンク付きコンテキスト ID 一致用にプログラムされている場合、このフィールドを 0b00 にプログラムする必要があります。

このフィールドが 0b00 以外の値に設定されているときの、ブレークポイント一致の詳細については、*セキュリティ状態の制御について*P. C10-66 を参照して下さい。

デバッグ イベントの生成 P. C3-40 も参照して下さい。

ビット [15:14]、セキュリティ拡張機能が実装されていない場合

予約、UNK/SBZP

バイトアドレス選択、ビット [8:5]

このフィールドを使用して、DBGBVR に保持されているワードアドレスのうち、特定のバイトに対してのみ一致または不一致の比較を行うことができます。このフィールドの動作は、次の条件にも依存します。

- *DBGBVR の意味*フィールドが、IVA 一致または不一致用にプログラムされている。
- v7 デバッグの場合、アドレス範囲マスクフィールドが 0b00000（マスクなし）にプログラムされている。
- CPSR.J ビットと CPSR.T ビットによって示される、プロセッサの命令セット状態

このフィールドの使用方法の詳細については、*IVA 一致または不一致時のバイトアドレス選択の動作* P. C10-55 を参照して下さい。

次のいずれかの条件が当てはまる場合、このフィールドを 0b1111 にプログラムする必要があります。

- DBGBVR の意味フィールド（ビット [22:20]）が、リンク付きまたはリンクなしコンテキスト ID 一致用にプログラムされている。
- v7 デバッグの場合、アドレス範囲マスクフィールド（ビット [28:24]）が、0b00000 以外の値にプログラムされている。

この操作を行わない場合、ブレークポイントデバッグ イベントの生成は予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b11 Match in any mode.

This field must be programmed to 0b11 if the DBGBVR meaning field, bits [22:20], is programmed for Linked Context ID match.

Breakpoint enable, bit [0]

This bit enables the BRP. The meaning of this bit is:

0 Breakpoint disabled

1 Breakpoint enabled.

A BRP never generates Breakpoint debug events when its DBGBCR is disabled.

The debug logic reset values of all bits of the DBGBCR are UNKNOWN.

Note

- In v6 Debug and v6.1 Debug, the Breakpoint enable bit of the DBGBCR is set to 0 on a debug logic reset, disabling the breakpoint.
- In v7 Debug, a debugger must ensure that DBGBCR[0] has a defined state before it programs DBGDSCR[15:14] to enable debug.

Byte address selection behavior on IVA match or mismatch

The DBGBVR is programmed with a word address. If you have programmed the BRP for Linked or Unlinked IVA match or mismatch, you can program the *Byte address select* field, DBGBCR[8:5], so that the breakpoint hits only if certain byte addresses are accessed. The exact interpretation depends on the processor instruction set state, as indicated by the CPSR.J and CPSR.T bits, and on the bottom two bits of the IVA. Table C10-11 on page C10-56 shows the operation of byte address masking using the DBGBCR[8:5] field.

Note

In the following cases, you must program DBGBCR[8:5] to 0b1111:

- if you program the BRP for Linked or Unlinked Context ID match
- in v7 Debug, if you program the BRP for linked or unlinked IVA match or mismatch with a nonzero Address range mask.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

特権モード制御、ビット [2:1]

このフィールドを使用して、プロセッサのモードについてブレークポイント一致を条件付きにできます。このフィールドの可能な値は次のとおりです。

0b00 ユーザ、システム、スーパバイザの各モードで一致が行われます。
この値は、v7 デバッグでのみサポートされます。

0b01 すべての特権モードで一致が行われます。

0b10 ユーザモードでのみ一致が行われます。

0b11 すべてのモードで一致が行われます。

DBGBVR の意味フィールド（ビット [22:20]）が、リンク付きコンテキスト ID 一致用にプログラムされている場合、このフィールドを 0b11 にプログラムする必要があります。

ブレークポイントイネーブル、ビット [0]

このビットは、BRP を可能にします。このビットの意味は次のとおりです。

0 ブレークポイントが不可能です。

1 ブレークポイントが可能です。

BRP の DBGBCR が不可能のとき、その BRP がブレークポイントデバッグ イベントを生成することはありません。

DBGBCR のすべてのビットについて、デバッグロジック リセット時の値は未知です。

注

- v6 デバッグと v6.1 デバッグでは、DBGBCR のブレークポイントイネーブル ビットはデバッグロジック リセット時に 0 にセットされ、ブレークポイントを不可能にします。
- v7 デバッグでは、デバッガは DBGDSCR[15:14] をプログラムしてデバッグを可能にする前に、DBGBCR[0] に状態が定義されていることを確認する必要があります。

IVA 一致または不一致時のバイトアドレス選択の動作

DBGBVR にはワードアドレスがプログラムされます。BRP をリンク付きまたはリンクなし IVA 一致または不一致用にプログラムした場合、*バイトアドレス選択*フィールド (DBGBCR[8:5]) をプログラムし、特定のバイトアドレスがアクセスされたときのみブレークポイントがヒットするようにできます。正確な解釈は、CPSR.J および CPSR.T ビットによって示されるプロセッサの命令セット状態と、IVA の下位 2 ビットに依存します。DBGBCR[8:5] フィールドを使用したバイトアドレス マスクの動作を、P. C10- 56 の表 C10- 11 に示します。

注

次の場合、DBGBCR[8:5] を 0b1111 にプログラムする必要があります。

- BRP を、リンク付きまたはリンクなしコンテキスト ID 一致用にプログラムする。
- v7 デバッグで、0 以外のアドレス範囲マスクを使用して、BRP をリンク付きまたはリンクなしの IVA 一致または不一致用にプログラムする。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C10-11 Effect of byte address selection on Breakpoint generation

			This BRP programmed for:	
Instruction set[a]	**Instruction address**[b]	**DBGBCR[8:5]**	**IVA match**	**IVA mismatch**
Any	Any address	0000	Miss	Hit
ARM	DBGBVR<31:2>:'00'	1111	Hit	Miss
		0000	Miss	Hit
		Any other value	UNPREDICTABLE	
	Any other address	xxxx	Miss	Hit
Thumb or ThumbEE	DBGBVR<31:2>:'00'	xx11	Hit	Miss
		xx10	UNPREDICTABLE	
		xx01	UNPREDICTABLE	
		xx00	Miss	Hit
	DBGBVR<31:2>:'10'	11xx	Hit	Miss
		10xx	UNPREDICTABLE	
		01xx	UNPREDICTABLE	
		00xx	Miss	Hit
	Any other address	xxxx	Miss	Hit
Jazelle	DBGBVR<31:2>:'00'	xxx1	Hit	Miss
		xxx0	Miss	Hit
	DBGBVR<31:2>:'01'	xx1x	Hit	Miss
		xx0x	Miss	Hit
	DBGBVR<31:2>:'10'	x1xx	Hit	Miss
		x0xx	Miss	Hit

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C10-11 ブレークポイント生成へのバイトアドレス選択の影響

			この BRP のプログラム対象	
命令セット[a]	**命令アドレス**[b]	**DBGBCR[8:5]**	**IVA 一致**	**IVA 不一致**
任意	任意のアドレス	0000	ミス	ヒット
ARM	DBGBVR<31:2>:'00'	1111	ヒット	ミス
		0000	ミス	ヒット
		他のすべての値	予測不能	
	他のすべてのアドレス	xxxx	ミス	ヒット
Thumb または ThumbEE	DBGBVR<31:2>:'00'	xx11	ヒット	ミス
		xx10	予測不能	
		xx01	予測不能	
		xx00	ミス	ヒット
	DBGBVR<31:2>:'10'	11xx	ヒット	ミス
		10xx	予測不能	
		01xx	予測不能	
		00xx	ミス	ヒット
	他のすべてのアドレス	xxxx	ミス	ヒット
Jazelle	DBGBVR<31:2>:'00'	xxx1	ヒット	ミス
		xxx0	ミス	ヒット
	DBGBVR<31:2>:'01'	xx1x	ヒット	ミス
		xx0x	ミス	ヒット
	DBGBVR<31:2>:'10'	x1xx	ヒット	ミス
		x0xx	ミス	ヒット

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C10-11 Effect of byte address selection on Breakpoint generation (continued)

Instruction set[a]	Instruction address[b]	DBGBCR[8:5]	This BRP programmed for: IVA match	IVA mismatch
Jazelle	DBGBVR<31:2>:'11'	1xxx	Hit	Miss
		0xxx	Miss	Hit
	Any other address	xxxx	Miss	Hit

a. As indicated by the CPSR.J and CPSR.T bits.
b. For more information see the Note that follows this table.

In a processor with a trivial implementation of the Jazelle extension, generation of Breakpoint debug events is UNPREDICTABLE, and the value of a subsequent read from DBGBCR[8:5] is UNKNOWN, if the value written to DBGBCR[8:5] has either DBGBCR[8] != DBGBCR[7], or DBGBCR[6] != DBGBCR[5]. For a description of the trivial implementation of the Jazelle extension see *Trivial implementation of the Jazelle extension* on page B1-81.

Note

- In Table C10-11 on page C10-56, the instruction address value is the address of the first unit of the instruction. For more information, including what happens when the BRP hits the address of a unit of the instruction other than the first unit, see *IVA comparisons and instruction length* on page C3-10.
- In the ARMv7-R profile, the value of the Instruction Endianness bit, SCTLR.IE, does not affect the generation of breakpoint debug events. For more information about instruction endianness, see *Instruction endianness* on page A3-8.

For examples of how to program a BRP using byte address selection see *IVA comparison programming examples* on page C3-12.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C10-11 ブレークポイント生成へのバイトアドレス選択の影響（続き）

命令セット[a]	命令アドレス[b]	DBGBCR[8:5]	この BRP のプログラム対象 IVA 一致	IVA 不一致
Jazelle	DBGBVR<31:2>:'11'	1xxx	ヒット	ミス
		0xxx	ミス	ヒット
	他のすべてのアドレス	xxxx	ミス	ヒット

a. CPSR.J および CPSR.T ビットにより示されます。
b. 詳細については、この表の後にある注を参照して下さい。

Jazelle 拡張機能のトリビアル実装を持つプロセッサでは、DBGBCR[8:5] に書き込まれる値において DBGBCR[8] != DBGBCR[7] または DBGBCR[6] != DBGBCR[5] の場合、ブレークポイントデバッグ イベントの生成は予測不能で、DBGBCR[8:5] から以後に読み出される値は未知です。Jazelle 拡張機能のトリビアル実装の説明については、*Jazelle 拡張機能のトリビアル実装* P. B1-81 を参照して下さい。

注

- P. C10-56 の表 C10-11 において、命令アドレスの値は、命令の最初のユニットのアドレスです。命令の最初以外のユニットのアドレスに BRP がヒットした場合の結果など、詳細については *IVA 比較と命令長* P. C3-10 を参照して下さい。
- ARMv7-R プロファイルでは、命令のエンディアン形式ビット (SCTLR.IE) の値は、ブレークポイントデバッグ イベントの生成に影響しません。命令のエンディアン形式の詳細については、*命令のエンディアン形式* P. A3-8 を参照して下さい。

バイトアドレス選択を使用して BRP をプログラムする方法の例については、*IVA 比較のプログラミング例* P. C3-12 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Summary of breakpoint generation options

Table C10-12 shows which values are compared and which are not for each type of BRP. Table entries in **bold typewriter** indicate an element of the comparison that is made. Reading across the *Comparison* columns for a row of the table gives the comparison to be made. For example, for the Linked IVA mismatch (0b001), the comparison is:

Not (**Equals**[IVA] AND **Set**[Byte lanes]) AND **Match**[State] AND **Link**[Linked Breakpoint]

Breakpoint generation is described by the `BRPMatch()` pseudocode function, see *Breakpoints and Vector Catches* on page C3-28.

Table C10-12 DBGBVR meaning bits summary

BRP type bits[a]	Description	Comparison				
		IVA[b]	Byte lanes[c]	Context ID[d]	State[e]	Linked
000	IVA match	**Equals**	AND **Set**		AND **Match**	
001	Linked IVA match[f]	**Equals**	AND **Set**		AND **Match**	AND **Link**
010	Context ID[g, h]			**Equals**	AND **Match**	
011	Linked Context ID[g, i]			**Equals**		AND **Link**
100	IVA mismatch[h]	Not (**Equals**	AND **Set**)		AND **Match**	
101	Linked IVA mismatch[f, h]	Not (**Equals**	AND **Set**)		AND **Match**	AND **Link**
11x	Reserved	-	-	-	-	-

a. The *DBGBVR meaning* field, DBGBCR[22:20].
b. Matching IVA[31:2] against DBGBVR[31:2]. If the breakpoint Address range mask bits [28:24] are set to a value other than 0b00000, a masked comparison is used. See *Breakpoint address range masking behavior, v7 Debug* on page C3-9.
c. IVA byte lanes. DBGBCR[8:5] indicate the byte lanes to be compared, see *Byte address selection behavior on IVA match or mismatch* on page C10-55.
d. Matching CONTEXTIDR[31:0] against DBGBVR[31:0].
e. Processor state comparison made, according to value of DBGBCR[15:14, 2:1], see *Breakpoint Control Registers (DBGBCR)* on page C10-49.
f. The Context ID is compared against the value of the linked breakpoint and a breakpoint event is only generated when both conditions match. If the linked breakpoint is not capable of Context ID comparison, or is not configured for Linked Context ID match, the generation of Breakpoint debug events is UNPREDICTABLE.
g. DBGBCR[8:5] for this BRP must be programmed to 0b1111; otherwise the generation of Breakpoint debug events is UNPREDICTABLE.
h. When Monitor debug-mode is selected, take care when programming DBGBCR[2:1], Privileged access control. See *Unpredictable behavior on Software debug events* on page C3-24 for more information.
i. See *Linked comparisons* on page C10-59.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ブレークポイント生成オプションの要約

BRP の各タイプについて、比較される値とされない値とを、表 C10-12 に示します。この表で、太字のタイプライタ書体のエントリは、行われる比較の要素を示しています。表の行について、*比較*列を横に読むと、行われる比較を知ることができます。例えば、リンク付き IVA 不一致 (0b001) の場合、比較は次のようになります。

Not (**Equals**[IVA] AND **Set**[バイトレーン]) AND **Match**[状態] AND **Link**[リンク付きブレークポイント]

ブレークポイントの生成は、`BRPMatch()` 擬似コード関数で記述されます。*ブレークポイントとベクタキャッチ*P. C3-28 を参照して下さい。

表 C10-12 DBGBVR の意味ビットの要約

BRP のタイプビット[a]	説明	比較				
		IVA[b]	バイトレーン[c]	コンテキスト ID[d]	状態[e]	リンク付き
000	IVA 一致	**Equals**	AND **Set**		AND **Match**	
001	リンク付きの IVA 一致[f]	**Equals**	AND **Set**		AND **Match**	AND **Link**
010	コンテキスト ID[g,h]			**Equals**	AND **Match**	
011	リンク付きコンテキスト ID[g,i]			**Equals**		AND **Link**
100	IVA 不一致[h]	Not (**Equals**	AND **Set**)		AND **Match**	
101	リンク付き IVA 不一致[f、h]	Not (**Equals**	AND **Set**)		AND **Match**	AND **Link**
11x	予約	–	–	–	–	–

a. *DBGBVR の意味*フィールド、DBGBCR[22:20]
b. IVA[31:2] と DBGBVR[31:2] との一致。ブレークポイントアドレス範囲マスクビット [28:24] が 0b00000 以外の値に設定されている場合、マスク付き比較が使用されます。*v7 デバッグでのブレークポイントアドレス範囲マスクの動作*P. C3-9 を参照して下さい。
c. IVA バイトレーン。DBGBCR[8:5] は、比較されるバイトレーンを示します。*IVA 一致または不一致時のバイトアドレス選択の動作*P. C10-55 を参照して下さい。
d. CONTEXTIDR[31:0] と DBGBVR[31:0] との一致
e. DBGBCR[15:14、2:1] の値に従って行われるプロセッサ状態の比較。*ブレークポイント制御レジスタ (DBGBCR)*P. C10-49 を参照して下さい。
f. コンテキスト ID がリンク付きブレークポイントの値と比較され、両方の条件が一致する場合のみブレークポイントイベントが生成されます。リンク付きブレークポイントがコンテキスト ID 比較の機能を持っていない場合、またはリンク付きコンテキスト ID 一致用に構成されていない場合、ブレークポイントデバッグ イベントの生成は予測不能です。
g. この BRP の DBGBCR[8:5] は 0b1111 にプログラムする必要があります。そうでない場合、ブレークポイントデバッグ イベントの生成は予測不能です。
h. モニタデバッグ モードが選択されているとき、DBGBCR[2:1]（特権アクセス制御）のプログラムには注意が必要です。詳細については、*ソフトウェアデバッグ イベントでの予測不能な動作*P. C3-24 を参照して下さい。
i. *リンク付き比較*P. C10-59 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Linked comparisons

For linked comparisons, a comparison includes a Context ID match, defined by a BRP, with one or more address comparisons defined by other BRPs or WRPs linked to the Context ID match:

- Zero or more other BRPs, each programmed to define a linked IVA match.
- Zero or more other BRPs, each programmed to define a linked IVA mismatch.

  Note

  Linked IVA mismatch is not supported in v6 Debug.

- Zero or more WRPs, each programmed to define a linked DVA match.

The Breakpoint or Watchpoint debug event is generated only if both:

- the Context ID match is true
- the IVA match or mismatch, or the DVA match, is true.

In this description:

- BRPm is used to define the Context ID match.
- BRPn is configured to define a linked IVA match or mismatch, and is linked to BRPm
- WRPn is configured to define a linked DVA match, and is linked to BRPm

If there are no BRPs and no WRPs linked to BRPm then BRPm cannot generate any debug events. The rest of this description assumes at least one BRP or WRP is linked to BRPm.

To configure BRPm to define the Context ID match part of the linked Context ID matches:

- program DBGBVRm[31:0] with the Context ID to be matched
- program DBGBCRm[22:20] to 0b011, linked Context ID comparison
- generation of the debug events is UNPREDICTABLE unless all of these conditions are met:
  - DBGBCRm[19:16] is programmed to 0b0000
  - DBGBCRm[15:14] is programmed to 0b00
  - DBGBCRm[8:5] is programmed to 0b1111
  - DBGBCRm[2:1] is programmed to 0b11.

To configure BRPn to define the IVA match or mismatch part of a linked Context ID match:

- program DBGBVRn[31:2] with the address for comparison, and DBGBVRn[1:0] to b00
- program DBGBCRn[22:20] to either:
  - 0b001, for linked IVA match
  - 0b101, for linked IVA mismatch, in v6.1 Debug or v7 Debug
- program DBGBCRn[19:16] to m, the number of the BRP that defines the Context ID match
- if required, program DBGBCRn[15:14,2:1] to include the state of the processor in the comparison.

To configure WRPn to define the DVA match part of a linked Context ID match:

- program DBGWVRn[31:2] with the address for comparison, and DBGWVRn[1:0] to b00
- program DBGWCRn[20] to 1, to enable linking
- program DBGWCRn[19:16] to m, the number of the BRP that defines the Context ID match
- if required, program DBGWCRn[15:14,2:1] to include the state of the processor in the comparison.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

リンク付き比較

リンク付き比較の場合、比較には BRP によって定義されるコンテキスト ID 一致と、コンテキスト ID 一致にリンクされている他の BRP または WRP によって定義される 1 つ以上のアドレス比較が含まれます。

- 0 個以上の他の BRP。それぞれ、リンク付き IVA 一致を定義するようにプログラムされます。
- 0 個以上の他の BRP。それぞれ、リンク付き IVA 不一致を定義するようにプログラムされます。

  注

  v6 デバッグでは、リンク付き IVA 不一致はサポートされていません。

- 0 個以上の他の WRP。それぞれ、リンク付き DVA 一致を定義するようにプログラムされます。

ブレークポイントまたはウォッチポイントデバッグ イベントは、次の両方が満たされる場合にのみ生成されます。

- コンテキスト ID 一致が TRUE である。
- IVA 一致または不一致、あるいは DVA 一致が TRUE である。

この説明では、次の前提を使用します。

- BRPm は、コンテキスト ID 一致を定義するために使用されます。
- BRPn は、リンク付き IVA 一致または不一致を定義するように構成され、BRPm にリンクされています。
- WRPn は、リンク付き DVA 一致を定義するように構成され、BRPm にリンクされています。

BRP が存在せず、BRPm に WRP がリンクされていない場合、BRPm はデバッグイベントを生成できません。以下の説明では、最低 1 つの BRP または WRP が BRPm にリンクされていることを前提とします。

リンク付きコンテキスト ID 一致のコンテキスト ID 一致部分を定義するように BRPm を構成するには、次のようにプログラムします。

- DBGBVRm[31:0] を、一致対象のコンテキスト ID にプログラムします。
- DBGBCRm[22:20] を、0b011（リンク付きコンテキスト ID 比較）にプログラムします。
- 次の条件がすべて満たされていない限り、デバッグ イベントの生成は予測不能です。
  - DBGBCRm[19:16] が 0b0000 にプログラムされている。
  - DBGBCRm[15:14] が 0b00 にプログラムされている。
  - DBGBCRm[8:5] が 0b1111 にプログラムされている。
  - DBGBCRm[2:1] が 0b11 にプログラムされている。

リンク付きコンテキスト ID 一致の IVA 一致または不一致部分を定義するように BRPn を構成するには、次のようにプログラムします。

- DBGBVRn[31:2] を比較対照のアドレスに、DBGBVRn[1:0] を b00 にプログラムします。
- DBGBCRn[22:20] を次のいずれかにプログラムします。
  - リンク付き IVA 一致の場合、0b001
  - リンク付き IVA 不一致の場合、0b101（v6.1 デバッグまたは v7 デバッグ）
- DBGBCRn[19:16] を m にプログラムします。m は、コンテキスト ID 一致を定義する BRP の番号です。
- 必要なら、プロセッサの状態を比較に含めるように DBGBCRn[15:14、2:1] をプログラムします。

リンク付きコンテキスト ID 一致の DVA 一致部分を定義するように WRPn を構成するには、次のようにプログラムします。

- DBGWVRn[31:2] を比較対照のアドレスに、DBGWVRn[1:0] を b00 にプログラムします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The generation of Breakpoint debug events is UNPREDICTABLE if:

- BRPn is linked to BRPm but is not configured for Linked IVA match or Linked IVA mismatch
- WRPn is linked to BRPm but is not configured to enable linking
- WRPn or BRPn is linked to BRPm and either:
  - BRPm does not support Linked Context ID matching
  - BRPm is not configured for Linked Context ID matching.

C10.5.3 Watchpoint Value Registers (DBGWVR)

A Watchpoint Value Register, DBGWVR, holds a *Data Virtual Address* (DVA) value for use in watchpoint matching. Each DBGWVR is associated with a DBGWCR to form a *Watchpoint Register Pair* (WRP). DBGWVRn is associated with DBGWCRn to form WRPn, where n takes the values from 0 to 15. A debug event is generated when the WRP is matched. For more information, see *Watchpoint debug events* on page C3-15.

A watchpoint can be set on either:

- a DVA match
- a DVA match occurring with a Context ID match.

For the second case:

- a WRP and a BRP with Context ID comparison capability have to be linked, see *Watchpoint Control Registers (DBGWCR)* on page C10-61 and *Linked comparisons* on page C10-59.
- a debug event is generated when, on the same instruction, both:
  - the DVA matches
  - the Context ID matches.

See *Memory addresses* on page C3-23 for a definition of the DVA used to program a DBGWVR.

The DBGWVRs are:

- debug registers 96-111, at offsets `0x180-0x1BC`
- read/write registers
- when the Security Extensions are implemented, Common registers.

The format of a DBGWVR is:

31 … 2	1	0
Watchpoint Address[31:2]	UNK/SBZP	

Bits [31:2] Bits [31:2] of the value for comparison, DVA[31:2].

Bits [1:0] Reserved. UNK/SBZP.

The debug logic reset value of a DBGWVR is UNKNOWN.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- DBGWCRn[20] を 1 にプログラムし、リンクを可能にします。
- DBGWCRn[19:16] を m にプログラムします。m は、コンテキスト ID 一致を定義する BRP の番号です。
- 必要なら、プロセッサの状態を比較に含めるように DBGWCRn[15:14、2:1] をプログラムします。

次の場合、ブレークポイントデバッグ イベントの生成は予測不能です。

- BRPn が BRPm にリンクされているが、リンク付き IVA 一致またはリンク付き IVA 不一致用に構成されていない。
- WRPn が BRPm にリンクされているが、リンクが可能になるよう構成されていない。
- WRPn または BRPn が BRPm にリンクされており、次のいずれかが当てはまる。
  - BRPm がリンク付きコンテキスト ID 一致をサポートしていない。
  - BRPm が、リンク付きコンテキスト ID 一致用に構成されていない。

C10.5.3 ウォッチポイント値レジスタ (DBGWVR)

ウォッチポイント値レジスタ (DBGWVR) は、ウォッチポイント一致に使用するための*データ仮想アドレス* (DVA) の値を保持します。それぞれの DBGWVR は DBGWCR と関連付けられ、*ウォッチポイントレジスタ ペア* (WRP) を構成します。DBGWVRn は DBGWCRn と関連付けられ、WRPn を構成します。ここで、n の値は 0 ～ 15 です。デバッグイベントは、WRP が一致したときに生成されます。詳細については、*ウォッチポイントデバッグ イベント* P. C3-15 を参照して下さい。

ウォッチポイントは、次のいずれかに設定できます。

- DVA 一致
- コンテキスト ID 一致とともに発生する DVA 一致

2 番目の場合、次の規則が適用されます。

- コンテキスト ID 比較機能を持つ BRP を WRP とリンクする必要があります。*ウォッチポイント制御レジスタ (DBGWCR)* P. C10-61 および*リンク付き比較* P. C10-59 を参照して下さい。
- デバッグイベントは、同じ命令で次の条件の両方が満たされたときに生成されます。
  - DVA 一致
  - コンテキスト ID 一致

DBGWVR をプログラムするために使用される DVA の定義については、*メモリアドレス* P. C3-23 を参照して下さい。

DBGWVR の特徴は次のとおりです。

- デバッグレジスタ 96 ～ 111、オフセット `0x180` ～ `0x1BC`
- 読み出し / 書き込みレジスタ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

DBGWVR のフォーマットは次のとおりです。

31 … 2	1	0
ウォッチポイントアドレス [31:2]	UNK/SBZP	

ビット [31:2] 比較する値のビット [31:2]、DVA[31:2]

ビット [1:0] 予約。UNK/SBZP

デバッグロジック リセット時の DBGWVR の値は未知です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.5.4 Watchpoint Control Registers (DBGWCR)

A Watchpoint Control Register, DBGWCR, holds control information for a watchpoint. Each DBGWCR is associated with a DBGWVR to form a *Watchpoint Register Pair* (WRP). For more information about WRPs and the possible watchpoints see *Watchpoint Value Registers (DBGWVR)* on page C10-60.

The DBGWCRs are:

- debug registers 112-127, at offsets `0x1C0-0x1FC`
- read/write registers
- when the Security Extensions are implemented, Common registers.

The format of a DBGWCR, in v7 Debug, is:

See the bit descriptions for the differences in other Debug architecture versions.

Bits [31:29, 23:21,13]

Reserved, UNK/SBZP.

Address range mask, bits [28:24], v7 Debug

In v7 Debug, support for watchpoint address range masking is optional. If it is not supported these bits are RAZ/WI.

If watchpoint address range masking is supported, this field can be used to watch a range of addresses by masking lower order address bits out of the watchpoint comparison. The value of this field is the number of low order bits of the address that are masked off, except that values of 1 and 2 are reserved. Therefore, the meaning of Watchpoint Address range mask values are:

0b00000 No mask
0b00001 Reserved
0b00010 Reserved
0b00011 `0x00000007` mask for data address, three bits masked
0b00100 `0x0000000F` mask for data address, four bits masked
0b00101 `0x0000001F` mask for data address, five bits masked
. .
. .
. .
0b11111 `0x7FFFFFFF` mask for data address, 31 bits masked.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.5.4 ウォッチポイント制御レジスタ (DBGWCR)

ウォッチポイント制御レジスタ (DBGWCR) は、ウォッチポイントの制御情報を保持します。それぞれの DBGWCR は DBGWVR と関連付けられ、*ウォッチポイントレジスタ ペア*(WRP) を構成します。WRP と可能なウォッチポイントの詳細については、*ウォッチポイント値レジスタ (DBGWVR)* P. C10- 60 を参照して下さい。

DBGWCR の特徴は次のとおりです。

- デバッグレジスタ 112 ～ 127、オフセット `0x1C0` ～ `0x1FC`
- 読み出し / 書き込みレジスタ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

v7 デバッグでの DBGWCR のフォーマットは次のとおりです。

デバッグアーキテクチャの他のバージョンにおける相違点については、ビットの説明を参照して下さい。

ビット [31:29、23:21、13]

予約、UNK/SBZP

アドレス範囲マスク、ビット [28:24]、v7 デバッグ

v7 デバッグでは、ウォッチポイントアドレス範囲マスクのサポートはオプションです。サポートされない場合、これらのビットは RAZ/WI です。

ウォッチポイントアドレス範囲マスクがサポートされている場合、このフィールドを使用して、ウォッチポイント比較アドレスの下位ビットをマスクにより除外し、一定範囲のアドレスにウォッチポイントを設定できます。このフィールドの値は、アドレスのうちマスクにより除外される下位ビットの数を示します。ただし、値 1 と 2 は予約されています。したがって、ウォッチポイントアドレス範囲マスクの値の意味は次のとおりです。

0b00000 マスクなし
0b00001 予約
0b00010 予約
0b00011 データアドレスに `0x00000007` マスクが適用され、3 ビットがマスクされます。
0b00100 データアドレスに `0x0000000F` マスクが適用され、4 ビットがマスクされます。
0b00101 データアドレスに `0x0000001F` マスクが適用され、5 ビットがマスクされます。
. .
. .
. .
0b11111 データアドレスに `0x7FFFFFFF` マスクが適用され、31 ビットがマスクされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

This field must be programmed to 0b00000 if either:

- DBGWCR[12:5] != 0b11111111, if an 8-bit Byte address select field is implemented
- DBGWCR[8:5] != 0b1111, if a 4-bit Byte address select field is implemented.

If this is not done, the generation of Watchpoint debug events is UNPREDICTABLE.

If this field is not zero, the DBGWVR bits that are not included in the comparison must be zero, otherwise the generation of Watchpoint debug events is UNPREDICTABLE.

To watch for a write to any byte in an doubleword-aligned object of size 8 bytes, ARM recommends that debuggers set DBGWCR[28:24] = 0x7, and DBGWCR[12:5] = 0b11111111. This setting is compatible with both implementations with an 8-bit Byte address select field and implementations with a 4-bit Byte address select field, because implementations with a 4-bit Byte address select field ignore writes to DBGWCR[12:9].

Bits [28:24], v6 Debug and v6.1 Debug

Reserved, UNK/SBZP.

Enable linking, bit [20]

This bit is set to 1 if this WRP is linked to a BRP to set a linked watchpoint that requires both DVA and Context ID comparison. The possible values of this bit are

0 linking disabled

1 linking enabled.

When this bit is set to 1 the Linked BRP number field indicates the BRP to which this WRP is linked. For more information, see *Linked comparisons* on page C10-59.

Linked BRP number, bits [19:16]

If this WRP is programmed with linking enabled then this field must be programmed with the number of the BRP that holds the Context ID to be used for the combined DVA and Context ID comparison, otherwise, this field must be programmed to 0b0000.

If this field is programmed with a value other than zero or the number of a BRP that supports Context ID comparison then reading this register returns an UNKNOWN value for this field.

The generation of Watchpoint debug events is UNPREDICTABLE if either:

- this WRP does not have linking enabled and this field is not programmed to 0b0000
- this WRP has linking enabled and the BRP indicated by this field does not support Context ID comparison or is not programmed for Linked Context ID match.

Security state control, bits [15:14], when the Security Extensions are implemented

Note

The Security Extensions cannot be implemented with v6 Debug.

When a processor implements the Security Extensions, these bits enable the breakpoint to be conditional on the security state of the processor:

0b00 watchpoint generated on match in both Non-secure state and Secure state

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

次のいずれかの条件が当てはまる場合、このフィールドを 0b00000 にプログラムする必要があります。

- 8 ビット バイトアドレス選択フィールドが実装されており、DBGWCR[12:5] != 0b11111111 の場合
- 4 ビット バイトアドレス選択フィールドが実装されており、DBGWCR[8:5] != 0b1111 の場合

この操作を行わない場合、ウォッチポイントデバッグ イベントの生成は予測不能です。

このフィールドが 0 以外のとき、比較に含まれない DBGWVR ビットは 0 の必要があります。そうでない場合、ウォッチポイントデバッグ イベントの生成は予測不能です。

ダブルワードアラインドでサイズが 8 バイトのオブジェクトに含まれる任意のバイトへの書き込みをウォッチするには、デバッガで DBGWCR[28:24] = 0x7 かつ DBGWCR[12:5] = 0b11111111 に設定することを推奨します。この設定は、8 ビット バイトアドレス選択フィールドと 4 ビット バイトアドレス選択フィールドの両方の実装に互換です。これは、4 ビット バイトアドレス選択フィールドの実装では、DBGWCR[12:9] への書き込みが無視されるためです。

ビット [28:24]、v6 デバッグおよび v6.1 デバッグのみ

予約、UNK/SBZP

リンクイネーブル、ビット [20]

このビットは、この WRP が BRP にリンクされ、DVA とコンテキスト ID 比較の両方を必要とするリンク付きウォッチポイントの設定に使用される場合に 1 にセットされます。このビットの可能な値は次のとおりです。

0 リンク付けは不可能です。

1 リンク付けが可能です。

このビットが 1 にセットされている場合、リンク付き BRP 番号フィールドは、この WRP がリンクされている先の BRP を示します。詳細については、*リンク付き比較* P. C10- 59 を参照して下さい。

リンク付き BRP 番号、ビット [19:16]

この WRP がリンク付け可能にプログラムされる場合、このフィールドには、DVA とコンテキスト ID の複合比較に使用されるコンテキスト ID を保持する BRP の番号をプログラムする必要があります。それ以外の場合、このフィールドを 0b0000 にプログラムする必要があります。

このフィールドが、0 またはコンテキスト ID 比較をサポートする BRP 番号のいずれでもない値にプログラムされている場合、このレジスタを読み出したときにこのフィールドについて返される値は未知です。

次のいずれかの場合、ウォッチポイントデバッグ イベントの生成は予測不能です。

- このWRPでリンク付けが可能でなく、このフィールドが 0b0000 にプログラムされていない。
- この WRP でリンク付けが可能で、このフィールドによって示される BRP がコンテキスト ID 比較をサポートしていないか、リンク付きコンテキスト ID 一致用にプログラムされていない。

セキュリティ状態制御、ビット [15:14]、セキュリティ拡張機能が実装されている場合

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b01 watchpoint generated on match only in Non-secure state

0b10 watchpoint generated on match only in Secure state

0b11 Reserved.

For more information about watchpoint matching when this field is set to a value other than 0b00, see *About security state control* on page C10-66.

Bits [15:14], when the Security Extensions are not implemented

Reserved, UNK/SBZP.

Bit [12:9], v6 Debug and v6.1 Debug

Reserved, UNK/SBZP.

Byte address select, bits [12:5] or bits [8:5]

The width of this field can depend on the ARM Debug architecture version:

v6 Debug and v6.1 Debug

The Byte address select field is always 4 bits, DBGWCR[8:5]

v7 Debug

It is IMPLEMENTATION DEFINED whether a 4-bit or an 8-bit Byte address select field is implemented:

- an 8-bit Byte address select field is DBGWCR[12:5]
- if a 4-bit Byte address select field is implemented then the Byte address select field is DBGWCR[8:5] and DBGWCR[12:9] is RAZ/WI.

DBGWVRs are programmed with word-aligned addresses. This field enables the watchpoint to hits only if certain byte addresses are accessed. The watchpoint hits if an access hits any byte being watched, even if:

- the access size is larger than the size of the region being watched
- the access is unaligned, and the base address of the access is not in the same word of memory as the address in the DBGWVR.

For details of the use of this field see *Byte address masking behavior on DVA match* on page C10-65.

If the Address range mask field is implemented and programmed to a value other than 0b00000, no mask, then this field must be programmed to:

- 0b1111, if a 4-bit Byte address select field is implemented.
- 0b11111111, an 8-bit Byte address select field is implemented.

If this is not done, the generation of Watchpoint debug events is UNPREDICTABLE.

Load/store access control, bits [4:3]

This field enables watchpoint matching conditional on the type of access being made. Possible values of this field are:

0b00 Reserved.

0b01 Match on any load, Load-Exclusive, or swap.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

v6 デバッグでは、セキュリティ拡張機能を実装できません。

プロセッサにセキュリティ拡張機能が実装されている場合、これらのビットにより、プロセッサのセキュリティ状態についてブレークポイントを条件付きにできます。

0b00 非セキュア状態とセキュア状態の両方で、一致によりウォッチポイントが生成されます。

0b01 非セキュア状態での一致でのみウォッチポイントが生成されます。

0b10 セキュア状態での一致でのみウォッチポイントが生成されます。

0b11 予約

このフィールドが 0b00 以外の値に設定されているときのウォッチポイント一致の詳細については、*セキュリティ状態の制御について*P. C10-66 を参照して下さい。

ビット [15:14]、セキュリティ拡張機能が実装されていない場合

予約、UNK/SBZP

ビット [12:9]、v6 デバッグおよび v6.1 デバッグのみ

予約、UNK/SBZP

バイトアドレス選択、ビット [12:5] またはビット [8:5]

このフィールドの幅は、ARM デバッグアーキテクチャのバージョンによって異なる可能性があります。

v6 デバッグおよび v6.1 デバッグ

バイトアドレス選択フィールドは常に 4 ビット (DBGWCR[8:5]) です。

v7 デバッグ

- 4 ビットと 8 ビットのどちらのバイトアドレス選択フィールドが実装されるかは実装定義です。
- 8 ビット バイトアドレス選択フィールドは DBGWCR[12:5] です。
- 4 ビット バイトアドレス選択フィールドが実装されている場合、バイトアドレス選択フィールドは DBGWCR[8:5] で、DBGWCR[12:9] は RAZ/WI です。

DBGWVR は、ワードアラインドのアドレスにプログラムされます。このフィールドを使用して、特定のバイトアドレスがアクセスされたときのみウォッチポイントがヒットするように設定できます。ウオッチポイントは、次の場合でも、ウォッチ対象の任意のバイトに対するアクセスにヒットします。

- アクセスのサイズが、ウォッチ対象の領域のサイズよりも大きい。
- アクセスがアンアラインドで、アクセスのベースアドレスが、DBGWVR のアドレスと同じメモリワードに含まれていない。

このフィールドの使用方法の詳細については、*DVA 一致時のバイトアドレス マスクの動作* P. C10-65 を参照して下さい。

アドレス範囲マスクフィールドが実装され、0b00000（マスクなし）以外の値にプログラムされている場合、このフィールドを次のようにプログラムする必要があります。

- 4 ビット バイトアドレス選択フィールドが実装されている場合、0b1111
- 8 ビット バイトアドレス選択フィールドが実装されている場合、0b11111111

この操作を行わない場合、ウォッチポイントデバッグ イベントの生成は予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

0b10 Match on any store, Store-Exclusive or swap.

0b11 Match on any either type of access.

If an implementation supports watchpoint generation by:

- a memory hint instruction, then that instruction is treated as generating a load access
- a cache maintenance operation, then that operation is treated as generating a store access.

Privileged mode control, bits [2:1]

This field enables watchpoint matching conditional on the mode of the processor. Possible values of this field are:

0b00 Reserved.

0b01 Match privileged accesses.

0b10 Match unprivileged accesses.

0b11 Match all accesses.

Note

- For all cases the match refers to the privilege of the access, not the mode of the processor. For example, if the watchpoint is configured to match privileged accesses only (0b01), and the processor executes an LDRT instruction in a privileged mode, the watchpoint does not match.
- Permitted values of this field are not identical to those for the DBGBCR. In the DBGBCR only, in v7 Debug, the value 0b00 is permitted.

Watchpoint enable, bit [0]

This bit enables the WRP. The meaning of this bit is:

0 Watchpoint disabled

1 Watchpoint enabled.

A WRP never generates Watchpoint debug events when its DBGWCR is disabled.

The debug logic reset values of all bits of the DBGWCR is UNKNOWN.

Note

- In v6 Debug and v6.1 Debug, the Watchpoint enable bit of the DBGWCR is set to 0 on a debug logic reset, disabling the watchpoint.
- In v7 Debug, a debugger must ensure that DBGWCR[0] has a defined state before it programs DBGDSCR[15:14] to enable debug.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ロード / ストアアクセス制御、ビット [4:3]

このフィールドは、行われるアクセスのタイプについてウォッチポイント一致を条件付きにするために使用できます。このフィールドの可能な値は次のとおりです。

0b00 予約

0b01 ロード、排他ロード、スワップのいずれかに一致します。

0b10 ストア、排他ストア、スワップのいずれかに一致します。

0b11 両方のタイプのアクセスすべてに一致します。

実装で次の理由によるウォッチポイントの生成がサポートされている場合、それぞれの項目に示されている規則が適用されます。

- メモリヒント命令による生成がサポートされている場合、ロードアクセスを生成する命令として扱われます。
- キャッシュ保守操作による生成がサポートされている場合、ストアアクセスを生成する操作として扱われます。

特権モード制御、ビット [2:1]

このフィールドを使用して、プロセッサのモードについてウォッチポイント一致を条件付きにできます。このフィールドの可能な値は次のとおりです。

0b00 予約

0b01 特権アクセスでの一致

0b10 非特権アクセスでの一致

0b11 すべてのアクセスでの一致

注

- すべての場合について、一致の条件となるのはアクセスの特権で、プロセッサのモードではありません。例えば、ウォッチポイントが特権アクセスにのみ一致するように構成されており (0b01)、プロセッサが特権モードで LDRT 命令を実行すると、ウォッチポイントは一致しません。
- このフィールドに許可される値は、DBGBCR のものと同じではありません。DBGBCR でのみ、v7 デバッグで値 0b00 が許可されます。

ウォッチポイントイネーブル、ビット [0]

このビットは、WRP を可能にします。このビットの意味は次のとおりです。

0 ウォッチポイントは不可能です。

1 ウォッチポイントは可能です。

WRP の DBGWCR が不可能なとき、その WRP がウォッチポイントデバッグ イベントを生成することはありません。

DBGWCR のすべてのビットについて、デバッグロジック リセット時の値は未知です。

注

- v6 デバッグと v6.1 デバッグでは、DBGWCR のウォッチポイントイネーブル ビットはデバッグロジック リセット時に 0 にセットされ、ウォッチポイントを不可能にします。
- v7 デバッグでは、デバッガは DBGDSCR[15:14] をプログラムしてデバッグを可能にする前に、DBGWCR[0] に状態が定義されていることを確認する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Byte address masking behavior on DVA match

For each WRP, the DBGWVR is programmed with a word-aligned address. The Byte address select bits of the DBGWCR can be programmed so that the watchpoint hits if only certain bits of the watched address are accessed:

- in all implementations, DBGWCR[8:5] can be programmed to enable the watchpoint to hit on any access to one or more of the bytes of the word addressed by the associated DBGWVR
- in a v7 Debug implementation that supports an 8-bit Byte address select field, DBGWCR[12:5] can be programmed to enable the watchpoint to hit on any access to one or more of the bytes of the doubleword addressed by the associated DBGWVR.

In all cases, a Watchpoint debug event is generated if an access hits any byte being watched, even if:

- the access size is larger than the size of the region being watched
- the access is unaligned, and the base address of the access is not in the word of memory addressed by DBGWVR.

Table C10-13 and Table C10-14 on page C10-66 show the meaning of the Byte address select values. Table C10-13 shows the values that can be programmed in any implementation.

Table C10-13 Byte address select values, word-aligned address

DBGWCR[12:5] value	Description
00000000	Watchpoint never hits
xxxxxxx1	Watchpoint hits if byte at address `DBGWVR<31:2>:'00'` is accessed
xxxxxx1x	Watchpoint hits if byte at address `DBGWVR<31:2>:'01'` is accessed
xxxxx1xx	Watchpoint hits if byte at address `DBGWVR<31:2>:'10'` is accessed
xxxx1xxx	Watchpoint hits if byte at address `DBGWVR<31:2>:'11'` is accessed

In v6 Debug and v6.1 Debug only a 4-bit Byte address select field is implemented and DBGWCR[12:9] is UNK/SBZP.

In v7 Debug, it is IMPLEMENTATION DEFINED whether an implementation supports a 4-bit or an 8-bit Byte address select field:

- If the processor implements a 4-bit Byte address select field, then DBGWCR[12:9] is RAZ/WI.
- If the processor implements an 8-bit Byte address select field, then DBGWCR[12:9] can also be programmed, and, for a given watchpoint register pair:
  - — DBGWVR can be programmed with a doubleword-aligned address, with DBGWVR[2] = 0. In this case DBGWCR[12:5] can be programmed to match any of the 8 bytes in that doubleword value.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DVA 一致時のバイトアドレス マスクの動作

各 WRP について、DBGWVR はワードアラインドのアドレスにプログラムされます。DBGWCR のバイトアドレス選択ビットをプログラムし、ウォッチ対象のアドレスにある特定のビットがアクセスされたときのみ、ウォッチポイントがヒットするようにできます。

- すべての実装について、DBGWCR[8:5] をプログラムし、関連付けられている DBGWVR によってアドレス指定されるワードのうち、1 つまたは複数のバイトへのアクセスすべてにウォッチポイントがヒットするようにできます。
- 8 ビット バイトアドレス選択フィールドをサポートする v7 デバッグ実装では、DBGWCR[12:5] のプログラムにより、関連付けられている DBGWVR によってアドレス指定されるダブルワードのうち、1 つまたは複数のバイトへのアクセスすべてにウォッチポイントがヒットするようにできます。

すべての場合について、ウォッチ対象の任意のバイトへのアクセスがヒットした場合、次の条件に当てはまる場合も含めてウォッチポイントデバッグ イベントが生成されます。

- アクセスのサイズが、ウォッチ対象の領域のサイズよりも大きい。
- アクセスがアンアラインドで、アクセスのベースアドレスが、DBGWVR のアドレスと同じメモリワードに含まれていない。

バイトアドレス選択の値の意味を、表 C10-13 と P. C10-66 の表 C10-14 に示します。
表 C10-13 に示されている値は、すべての実装でプログラム可能です。

表 C10-13 バイトアドレス選択の値、ワードアラインド アクセス

DBGWCR [12:5] の値	説明
00000000	ウォッチポイントは一切ヒットしません。
xxxxxxx1	アドレス `DBGWVR<31:2>:'00'` のバイトにアクセスするとウォッチポイントがヒットします。
xxxxxx1x	アドレス `DBGWVR<31:2>:'01'` のバイトにアクセスするとウォッチポイントがヒットします。
xxxxx1xx	アドレス `DBGWVR<31:2>:'10'` のバイトにアクセスするとウォッチポイントがヒットします。
xxxx1xxx	アドレス `DBGWVR<31:2>:'11'` のバイトにアクセスするとウォッチポイントがヒットします。

v6 デバッグと v6.1 デバッグでは、4 ビットのバイトアドレス選択フィールドのみが実装され、DBGWCR[12:9] は UNK/SBZP です。

v7 デバッグでは、実装で 4 ビットまたは 8 ビットのバイトアドレス選択フィールドをサポートするかどうかは実装定義です。

- プロセッサに 4 ビットのバイトアドレス選択フィールドが実装されている場合、DBGWCR[12:9] は RAZ/WI です。
- プロセッサに8ビットのバイトアドレス選択フィールドが実装されている場合、DBGWCR[12:9] もプログラム可能で、与えられたウォッチポイント レジスタペアについて次の規則が適用されます。
  - – DBGWVR[2] = 0 を指定し、DBGWVR にダブルワードアラインドのアドレスをプログラムできます。この場合、DBGWCR[12:5] はダブルワード値の任意の 8 バイトに一致するようプログラム可能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

— If DBGWVR[2] == 1, indicating a word-aligned address that is not doubleword-aligned, then DBGWCR[12:9] must be programmed with zero.

If DBGWVR[2] == 1 and DBGWCR[12:9] != 0b0000, Watchpoint debug event generation is UNPREDICTABLE.

Table C10-14 shows the additional Byte address select field encodings that are available, when DBGWVR[2] == 0, on an implementation that supports an 8-bit Byte address select field.

Table C10-14 Additional Byte address select values, doubleword-aligned address

DBGWCR[12:5] value	Description
xxx1xxxx	Watchpoint hits if byte at address `DBGWVR<31:3>:'100'` is accessed
xx1xxxxx	Watchpoint hits if byte at address `DBGWVR<31:3>:'101'` is accessed
x1xxxxxx	Watchpoint hits if byte at address `DBGWVR<31:3>:'110'` is accessed
1xxxxxxx	Watchpoint hits if byte at address `DBGWVR<31:3>:'111'` is accessed

The same programming model can be used on implementations that support:

- an 8-bit Byte address select field, DBGWCR[12:5]
- a 4-bit Byte address select field, DBGWCR[8:5].

This is because, on an implementation that supports only a 4-bit Byte address select field, writes to DBGWCR[12:9] are ignored.

Note

In ARMv6, when using the optional legacy BE-32 endianness model, the values of DBGWCR[8:5] shown in Table C10-13 on page C10-65 have different meanings. For more information see *BE-32 DBGWCR Byte address select values* on page AppxG-7.

About security state control

When the Security Extensions are implemented and the security state control bits of the DBGBCR or DBGWCR are set to a value other than 0b00, the condition for matching refers to the security state of the processor, not the security of the access. For example, the breakpoint or watchpoint does not match when all of the following apply:

- the breakpoint or watchpoint is configured to match in Non-secure state only (0b01)
- the processor is executing code in the Secure state, either because the SCR.NS bit is 0 or because the processor is in Monitor mode
- the address accessed is in a page marked as Non-secure in the translation tables.

For more information about the security of accesses see Chapter B3 *Virtual Memory System Architecture (VMSA)*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

— If DBGWVR[2] == 1 の場合、ダブルワードアラインドでないワードアラウンドのアドレスが指定されており、DBGWCR[12:9] を 0 にプログラムする必要があります。

DBGWVR[2] == 1 かつ DBGWCR[12:9] != 0b0000 の場合、ウォッチポイントデバッグイベントの生成は予測不能です。

8 ビットのバイトアドレス選択フィールドをサポートする実装で、DBGWVR[2] == 0 のときに使用できる追加のバイトアドレス選択フィールドのエンコードを、表 C10-14 に示します。

表 C10-14 追加のバイトアドレス選択の値、ダブルワードアラインド アクセス

DBGWCR [12:5] の値	説明
xxx1xxxx	アドレス `DBGWVR<31:3>:'100'` のバイトにアクセスするとウォッチポイントがヒットします。
xx1xxxxx	アドレス `DBGWVR<31:3>:'101'` のバイトにアクセスするとウォッチポイントがヒットします。
x1xxxxxx	アドレス `DBGWVR<31:3>:'110'` のバイトにアクセスするとウォッチポイントがヒットします。
1xxxxxxx	アドレス `DBGWVR<31:3>:'111'` のバイトにアクセスするとウォッチポイントがヒットします。

次の形式をサポートする実装では、同じプログラミングモデルが使用できます。

- 8 ビットのバイトアドレス選択フィールド、DBGWCR[12:5]
- 4 ビットのバイトアドレス選択フィールド、DBGWCR[8:5]

これは、4 ビットのバイトアドレス選択フィールドのみをサポートする実装では、DBGWCR[12:9] が無視されるためです。

注

ARMv6 では、オプションの以前の BE-32 エンディアン形式モデルを使用する場合、P. C10-65 の表 C10-13 に示されている DBGWCR[8:5] の値の意味が異なります。詳細については、*BE-32 DBGWCR のバイトアドレス選択の値* P. 付録 G-7 を参照して下さい。

セキュリティ状態の制御について

セキュリティ拡張機能が実装されており、DBGBCR または DBGWCR のセキュリティ状態制御ビットが 0b00 以外の値に設定されている場合、一致の条件はアクセスのセキュリティではなく、プロセッサのセキュリティ状態を指すようになります。例えば、次のすべての条件が当てはまる場合、ブレークポイントまたはウォッチポイントは一致しません。

- ブレークポイントまたはウォッチポイントが、非セキュア状態でのみ一致するよう構成されている (0b01)。
- SCR.NS ビットが 0 であるか、プロセッサがモニタモードであるため、プロセッサがセキュア状態でコードを実行している。
- アクセスされたアドレスが、変換テーブルで非セキュアにマークされたページに含まれている。

アクセスのセキュリティの詳細については、第 B3 章*「仮想メモリシステム アーキテクチャ (VMSA)」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

This describes a VMSA access. In ARMv7 the Security Extensions can be implemented only in a system that implements the VMSA.

C10.5.5 Vector Catch Register (DBGVCR)

The Vector Catch Register, DBGVCR, enables Vector Catch debug events. A Vector Catch debug event occurs when:

- a bit in the DBGVCR is set to 1, to enable catches on a particular exception vector
- an IVA matches the corresponding exception vector address
- the instruction is committed for execution.

For more information, see *Vector Catch debug events* on page C3-20.

The DBGVCR is:

- debug register 7, at offset `0x01C`.
- a read/write register
- when the Security Extensions are implemented, a Common register.

The format the DBGVCR depends on whether the Security Extensions are implemented, and can depend on the ARM Debug architecture version:

Security Extensions not implemented

Only DBGVCR bits [7:6,4:0] are implemented. All other bits are reserved and UNK/SBZP.

Note

The Security Extensions cannot be implemented with v6 Debug.

v6.1 Debug When the Security Extensions are implemented it is optional whether DBGVCR bits [31,30,28:25,15,14,12:10] are implemented. If these bits are not implemented, they are RAZ/WI. However, for forwards compatibility with v7 Debug, ARM recommends that these bits are implemented.

DBGVCR bits [7:6,4:0] are always implemented. All other bits are reserved and UNK/SBZP.

v7 Debug When the Security Extensions are implemented DBGVCR bits [31,30,28:25,15,14,12:10] must be implemented.

DBGVCR bits [7:6,4:0] are also implemented. All other bits are reserved and UNK/SBZP.

When the Security Extensions are not implemented, and in any v6.1 Debug implementation that does not implement DBGVCR[31,30,28:25,15,14,12:10], the format of the DBGVCR is:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

これは VMSA アクセスの説明です。ARMv7 では、VMSA を実装しているシステムにのみセキュリティ拡張機能を実装できます。

C10.5.5 ベクタキャッチ レジスタ (DBGVCR)

ベクタキャッチ レジスタ (DBGVCR) は、ベクタキャッチデバッグ イベントを可能にします。ベクタキャッチデバッグ イベントは、次の場合に発生します。

- DBGVCR のビットが 1 にセットされ、特定の例外ベクタについてキャッチが可能になっている。
- IVA が、対応する例外ベクタアドレスに一致する。
- 命令が実行にコミットされている。

詳細については、*ベクタキャッチデバッグ イベント* P. C3-20 を参照して下さい。

DBGVCR の特徴は次のとおりです。

- デバッグレジスタ 7、オフセット `0x01C`
- 読み出し / 書き込みレジスタ
- セキュリティ拡張機能が実装されている場合、共通レジスタである。

DBGVCR のフォーマットは、セキュリティ拡張機能が実装されているかどうか、および ARM デバッグアーキテクチャのバージョンによって異なります。

セキュリティ拡張機能が実装されていない場合

DBGVCR のビット [7:6、4:0] のみが実装されています。他のビットはすべて予約されており、UNK/SBZP です。

注

v6 デバッグでは、セキュリティ拡張機能を実装できません。

v6.1 デバッグ セキュリティ拡張機能が実装されている場合、DBGVCR のビット [31、30、28:25、15、14、12:10] が実装されるかどうかはオプションです。これらのビットが実装されていない場合、RAZ/WI です。ただし、v7 デバッグとの上位互換性のため、これらのビットを実装することを推奨します。

DBGVCR のビット [7:6、4:0] は常に実装されます。他のビットはすべて予約されており、UNK/SBZP です。

v7 デバッグ セキュリティ拡張機能が実装されている場合、DBGVCR のビット [31、30、28:25、15、14、12:10] を必ず実装する必要があります。

DBGVCR のビット [7:6、4:0] も実装されます。他のビットはすべて予約されており、UNK/SBZP です。

セキュリティ拡張機能が実装されておらず、DBGVCR[31、30、28:25、15、14、12:10] を実装していない v6.1 デバッグ実装では、DBGVCR のフォーマットは次のとおりです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Bits [31:8,5] Reserved. Normally UNK/SBZP, except that in v6.1 Debug bits [31,30,28:25,15,14,12:10] are RAZ/WI, and the other bits are UNK/SBZP.

Bits [7:6,4:0] Vector catch enable bits. When one of these bits is set to 1, any instruction prefetched from the corresponding exception vector generates a Vector Catch debug event when it is committed for execution. Table C10-15 on page C10-71 shows the exception vectors. The Vector Catch enable bits are:

- **Bit [7]** FIQ vector catch enable.
- **Bit [6]** IRQ vector catch enable.
- **Bit [4]** Data Abort vector catch enable.
- **Bit [3]** Prefetch Abort vector catch enable.
- **Bit [2]** SVC vector catch enable.
- **Bit [1]** Undefined Instruction vector catch enable.
- **Bit [0]** Reset vector catch enable.

The debug logic reset value of the DBGVCR depends on the ARM Debug architecture version:

v7 Debug Debug logic reset values are UNKNOWN. Before programming DBGDSCR[15:14] to enable debug, a debugger must ensure that the DBGVCR has a defined state.

v6 Debug and v6.1 Debug

All defined bits reset to 0.

If Monitor debug-mode is configured and enabled DBGVCR bits [4:3] must be programmed to 0b00, see *Unpredictable behavior on Software debug events* on page C3-24

For more information about these vector catch operations see *Vector catch operation when Security Extensions are not implemented* on page C10-71.

When the Security Extensions are implemented the format of the DBGVCR is:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ビット[31:8、5] 予約。通常は UNK/SBZP です。ただし v6.1 デバッグではビット [31、30、28:25、15、14、12:10] は RAZ/WI、他のビットは UNK/SBZP です。

ビット[7:6、4:0] ベクタキャッチイネーブル ビット。これらのビットの 1 つが 1 にセットされていると、対応する例外ベクタからプリフェッチされた命令は、実行にコミットされたときにベクタキャッチデバッグ イベントを生成します。例外ベクタを P. C10-71 の表 C10-15 に示します。ベクタキャッチイネーブル ビットは次のとおりです。

- **ビット[7]** FIQ ベクタキャッチ イネーブル
- **ビット[6]** IRQ ベクタキャッチ イネーブル
- **ビット[4]** データアボート ベクタキャッチイネーブル
- **ビット[3]** プリフェッチアボート ベクタキャッチイネーブル
- **ビット[2]** SVC ベクタキャッチ イネーブル
- **ビット[1]** 未定義命令ベクタキャッチ イネーブル
- **ビット[0]** リセットベクタキャッチ イネーブル

デバッグロジック リセット時の DBGVCR の値は、ARM デバッグアーキテクチャのバージョンによって異なります。

v7 デバッグ デバッグロジック リセット時の値は未知です。デバッガは、DBGDSCR[15:14] をプログラムしてデバッグを可能にする前に、DBGVCR が定義済みの状態であることを確認する必要があります。

v6 デバッグと v6.1 デバッグ

定義済みのビットはすべて 0 にリセットされます。

モニタデバッグ モードが構成されていて可能な場合、DBGVCR のビット [4:3] を 0b00 にプログラムする必要があります。*ソフトウェアデバッグ イベントでの予測不能な動作* P. C3-24 を参照して下さい。

これらのベクタキャッチ操作の詳細については、*セキュリティ拡張機能が実装されていない場合のベクタキャッチ操作* P. C10-71 を参照して下さい。

セキュリティ拡張機能が実装されている場合の DBGVCR のフォーマットは次のとおりです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Bits [29,24:14,13,9:8,5]

Reserved. UNK/SBZP.

Bits [31:30,28:25]

Vector catch enable bits for exceptions in the Non-secure state. When one of these bits is set to 1 and the processor is in the Non-secure state, a Vector Catch debug event is generated when an instruction prefetched from the corresponding exception vector is committed for execution. Table C10-16 on page C10-73 shows the exception vectors. The Non-secure state vector catch enable bits are:

Bit [31] FIQ vector catch enable in Non-secure state.

Bit [30] IRQ vector catch enable in Non-secure state.

Bit [28] Data Abort vector catch enable in Non-secure state.

Bit [27] Prefetch Abort vector catch enable in Non-secure state.

Bit [26] SVC vector catch enable in Non-secure state.

Bit [25] Undefined Instruction vector catch enable in Non-secure state.

Bits [15:14,12:10]

Vector catch enable bits for exceptions in the Secure state that are taken on the Monitor mode exception vectors. When one of these bits is set to 1 and the processor is in the Secure state, a Vector Catch debug event is generated when an instruction prefetched from the corresponding exception vector is committed for execution. Table C10-16 on page C10-73 shows the exception vectors. The Monitor mode vector catch enable bits are:

Bit [15] FIQ vector catch enable, in Secure state on Monitor mode vector.

Bit [14] IRQ vector catch enable in Secure state on Monitor mode vector.

Bit [12] Data Abort vector catch enable in Secure state on Monitor mode vector.

Bit [11] Prefetch Abort vector catch enable in Secure state on Monitor mode vector.

Bit [10] SMC vector catch enable in Secure state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ビット [29、24:14、13、9:8、5]

予約。UNK/SBZP

ビット [31:30、28:25]

非セキュア状態での例外についてのベクタキャッチイネーブル ビット。これらのビットの 1 つが 1 にセットされており、プロセッサが非セキュア状態のとき、対応する例外ベクタからプリフェッチされた命令が実行にコミットされたときに、ベクタキャッチデバッグ イベントが生成されます。例外ベクタを P. C10- 73 の表 C10- 16 に示します。非セキュア状態のベクタキャッチイネーブル ビットは次のとおりです。

ビット[31] 非セキュア状態の FIQ ベクタキャッチ イネーブル

ビット[30] 非セキュア状態の IRQ ベクタキャッチ イネーブル

ビット[28] 非セキュア状態のデータアボート ベクタキャッチイネーブル

ビット[27] 非セキュア状態のプリフェッチアボート ベクタキャッチイネーブル

ビット[26] 非セキュア状態の SVC ベクタキャッチ イネーブル

ビット[25] 非セキュア状態の未定義命令ベクタキャッチ イネーブル

ビット [15:14、12:10]

セキュア状態で、モニタモード例外ベクタについて取得された例外のベクタキャッチイネーブル ビット。これらのビットの 1 つが 1 にセットされており、プロセッサがセキュア状態のとき、対応する例外ベクタからプリフェッチされた命令が実行にコミットされたときに、ベクタキャッチデバッグ イベントが生成されます。例外ベクタを P. C10- 73 の表 C10- 16 に示します。モニタモードのベクタキャッチイネーブル ビットは次のとおりです。

ビット[15] FIQ ベクタキャッチ イネーブル（セキュア状態でのモニタモード ベクタ）

ビット[14] IRQ ベクタキャッチ イネーブル（セキュア状態でのモニタモード ベクタ）

ビット[12] データアボート ベクタキャッチイネーブル（セキュア状態でのモニタモード ベクタ）

ビット[11] プリフェッチアボート ベクタキャッチイネーブル（セキュア状態でのモニタモード ベクタ）

ビット[10] セキュア状態の SVC ベクタキャッチ イネーブル

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Bits [7:6,4:1] Vector catch enable bits for exceptions in the Secure state that are taken on the exception mode vector. When one of these bits is set to 1 and the processor is in the Secure state, a Vector Catch debug event is generated when an instruction prefetched from the corresponding exception vector is committed for execution. Table C10-16 on page C10-73 shows the exception vectors. The Secure state vector catch enable bits are:

Bit [7] FIQ vector catch enable in Secure state.

Bit [6] IRQ vector catch enable in Secure state.

Bit [4] Data Abort vector catch enable in Secure state.

Bit [3] Prefetch Abort vector catch enable in Secure state.

Bit [2] SVC vector catch enable in Secure state.

Bit [1] Undefined Instruction vector catch enable in Secure state.

Bit [0] Reset vector catch enable.

When this bit is set to 1, a Vector Catch debug event is generated when an instruction prefetched from the reset exception vector is committed for execution:

- In v7 Debug the debug event is generated regardless of the security state of the processor
- In v6 Debug and v6.1 Debug the debug event is only generated if the processor is in Secure state.

Table C10-16 on page C10-73 shows the exception vectors.

The debug logic reset value of the DBGVCR depends on the ARM Debug architecture version:

v7 Debug Debug logic reset values are UNKNOWN. Before programming DBGDSCR[15:14] to enable debug, a debugger must ensure that the DBGVCR has a defined state.

v6 Debug and v6.1 Debug

All defined bits reset to 0.

If Monitor debug-mode is configured and enabled DBGVCR bits [28:27,12,4:3] must be programmed to zero, see *Unpredictable behavior on Software debug events* on page C3-24

For more information about these vector catch operations see *Vector catch operation when Security Extensions are implemented* on page C10-71.

Vector catch operation

The following subsections give more information about vector catch operation:

- *Vector catch operation when Security Extensions are not implemented* on page C10-71
- *Vector catch operation when Security Extensions are implemented* on page C10-71.

The pseudocode function `VCRMatch()` describes the vector catch operation, for both the Secure and the Non-secure cases, and the function `VCR_OnTakingInterrupt()` tracks the most recent interrupt vectors. For more information about these pseudocode functions and when they are called see *Breakpoints and Vector Catches* on page C3-28.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ビット [7:6、4:1]

セキュア状態で、例外モードベクタについて取得された例外のベクタキャッチイネーブル ビット。これらのビットの 1 つが 1 にセットされており、プロセッサがセキュア状態のとき、対応する例外ベクタからプリフェッチされた命令が実行にコミットされたときに、ベクタキャッチデバッグ イベントが生成されます。例外ベクタを P. C10-73 の表 C10-16 に示します。セキュア状態のベクタキャッチイネーブル ビットは次のとおりです。

ビット [7] セキュア状態の FIQ ベクタキャッチ イネーブル

ビット [6] セキュア状態の IRQ ベクタキャッチ イネーブル

ビット [4] セキュア状態のデータアボート ベクタキャッチイネーブル

ビット [3] セキュア状態のプリフェッチアボート ベクタキャッチイネーブル

ビット [2] セキュア状態の SVC ベクタキャッチ イネーブル

ビット [1] セキュア状態の未定義命令ベクタキャッチ イネーブル

ビット [0] リセットベクタキャッチ イネーブル

このビットが 1 にセットされているとき、リセット例外ベクタからプリフェッチされた命令が実行にコミットされると、ベクタキャッチデバッグ イベントが生成されます。

- v7 デバッグでは、プロセッサのセキュリティ状態にかかわらずデバッグイベントが生成されます。
- v6 デバッグと v6.1 デバッグでは、プロセッサがセキュア状態の場合のみデバッグイベントが生成されます。

例外ベクタを P. C10-73 の表 C10-16 に示します。

デバッグロジックリセット時のDBGVCRの値は、ARM デバッグアーキテクチャのバージョンによって異なります。

v7 デバッグ デバッグロジック リセット時の値は未知です。デバッガは、DBGDSCR[15:14] をプログラムしてデバッグを可能にする前に、DBGVCR が定義済みの状態であることを確認する必要があります。

v6 デバッグと v6.1 デバッグ

定義済みのビットはすべて 0 にリセットされます。

モニタデバッグ モードが構成されており可能な場合、DBGVCR のビット [28:27、12、4:3] を 0 にプログラムする必要があります。*ソフトウェアデバッグ イベントでの予測不能な動作* P. C3-24 を参照して下さい。

これらのベクタキャッチ操作の詳細については、*セキュリティ拡張機能が実装されている場合のベクタキャッチ操作* P. C10-71 を参照して下さい。

ベクタキャッチ操作

次のサブセクションでは、ベクタキャッチ操作の詳細について説明します。

- *セキュリティ拡張機能が実装されていない場合のベクタキャッチ操作* P. C10-71
- *セキュリティ拡張機能が実装されている場合のベクタキャッチ操作* P. C10-71.

擬似関数 `VCRMatch()` は、セキュアおよび非セキュア両方の場合について、ベクタキャッチ操作を説明します。また、関数 `VCR_OnTakingInterrupt()` は最も新しい割り込みベクタを追跡します。これらの擬似コード関数と、どのような場合に呼び出されるかの詳細については、*ブレークポイントとベクタキャッチ* P. C3-28 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Vector catch operation when Security Extensions are not implemented

For each bit of the DBGVCR, the vector addresses caught depends on the exception vector configuration in the SCTLR:

- whether the SCTLR.V bit is programmed for Normal or High exception vectors
- for catches on the FIQ and IRQ exception vectors, on the programming of the SCTLR.VE bit.

Table C10-15 shows how the vector address that corresponds to each active bit of the DBGVCR depends on these configuration settings:

Table C10-15 Vector catch addresses, for processors without Security Extensions

DBGVCR bit	Vector catch enabled		Configured exception vectors	
			Normal (V == 0)	High (V == 1)
[7]	FIQ	VE == 0	0x0000001C	0xFFFF001C
		VE == 1	Most recent FIQ address[a]	
[6]	IRQ	VE == 0	0x00000018	0xFFFF0018
		VE == 1	Most recent IRQ address[a]	
[4]	Data Abort		0x00000010	0xFFFF0010
[3]	Prefetch Abort		0x0000000C	0xFFFF000C
[2]	SVC		0x00000008	0xFFFF0008
[1]	Undefined Instruction		0x00000004	0xFFFF0004
[0]	Reset		0x00000000	0xFFFF0000

a. For more information see *Vector catch debug events and vectored interrupt support* on page C3-22.

Vector catch operation when Security Extensions are implemented

When the Security Extensions are implemented, for each bit of the DBGVCR, the vector addresses caught depends:

- On the value programmed in the appropriate Vector Base Address Register:
  - — the Non-secure copy of the Vector Base Address Register ($VBAR_{NS}$) for the Non-secure state vector catches
  - — the Monitor Vector Base Address Register (MVBAR) for the Secure state vector catches on the Monitor mode vectors
  - — the Secure copy of the Vector Base Address Register ($VBAR_S$) for the Secure state vector catches on the exception vectors.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

セキュリティ拡張機能が実装されていない場合のベクタキャッチ操作

DBGVCR の各ビットについて、キャッチされるベクタアドレスは、SCTLR の例外ベクタの構成に依存します。

- SCTLR.V ビットが通常例外ベクタと上位例外ベクタのどちらにプログラムされているか
- FIQ および IRQ 例外ベクタのキャッチの場合、SCTLR.VE ビットのプログラム

DBGVCR のアクティブな各ビットに対応するベクタアドレスが、これらの構成設定にどのような影響を受けるかを、表 C10-15 に示します。

表 C10-15 ベクタキャッチ アドレス、セキュリティ拡張機能が実装されていないプロセッサ

DBGVCR のビット	可能なベクタキャッチ		構成されている例外ベクタ	
			通常 (V = 0)	上位 (V = 1)
[7]	FIQ	VE == 0	0x0000001C	0xFFFF001C
		VE == 1	最も新しい FIQ アドレス[a]	
[6]	IRQ	VE == 0	0x00000018	0xFFFF0018
		VE == 1	最も新しい IRQ アドレス[a]	
[4]	データアボート		0x00000010	0xFFFF0010
[3]	プリフェッチアボート		0x0000000C	0xFFFF000C
[2]	SVC		0x00000008	0xFFFF0008
[1]	未定義命令		0x00000004	0xFFFF0004
[0]	リセット		0x00000000	0xFFFF0000

a. 詳細については、*ベクタキャッチデバッグ イベントとベクタ割り込みのサポート* P. C3-22 を参照して下さい。

セキュリティ拡張機能が実装されている場合のベクタキャッチ操作

セキュリティ拡張機能が実装されている場合、DBGVCR の各ビットについて、キャッチされるベクタアドレスは次の要因に依存します。

- 対応するベクタベースアドレス レジスタにプログラムされている値
  - – 非セキュア状態でのベクタキャッチの場合、ベクタベースアドレス レジスタ ($VBAR_{NS}$) の非セキュアコピー
  - – モニタモード ベクタのセキュア状態でのベクタキャッチの場合、モニタ ベクタ ベースアドレス レジスタ (MVBAR)
  - – 例外ベクタのセキュア状態でのベクタキャッチの場合、ベクタベースアドレス レジスタ ($VBAR_S$) のセキュアコピー

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For more information about these registers see:

— *c12, Vector Base Address Register (VBAR)* on page B3-148

— *c12, Monitor Vector Base Address Register (MVBAR)* on page B3-149.

Note

The Reset exception vectors address never depends on the Vector Base Address values, and when SCTLR.VE == 1 the IRQ and FIQ exception vector addresses do not depend on the Vector Base Address values, see Table C10-16 on page C10-73 for more information.

- Except for the Secure state vector catches on the Monitor mode vectors, on the exception vector configuration in the SCTLR:
  — whether the SCTLR.V bit is programmed for Normal or High exception vectors
  — for catches on the FIQ and IRQ exception vectors, on the programming of the SCTLR.VE bit.

Generation of Vector Catch debug events also depends on the security state of the processor:

- the Non-secure state vector catches are generated only in Non-secure state
- the Secure state vector catches are generated only in Secure state
- in v6 Debug and v6.1 Debug, Reset vector catches are generated only in Secure state.

In v7 Debug, if Reset vector catch is enabled the Reset vector catches are generated regardless of the security state of the processor.

Generation of Vector Catch debug events takes no account of the values in the Secure Configuration Register (SCR), except for SCR.NS. For example, if the DBGVCR is programmed to catch Secure state IRQs on the Monitor mode vector, by setting bit [14] of the DBGVCR to 1, and the processor is in the Secure state, a Vector Catch debug event is generated on any instruction prefetch from (MVBAR + `0x18`). This debug event is generated even if the SCR is programmed for IRQs to be handled in IRQ mode.

Table C10-15 on page C10-71 shows, for each active bit of the DBGVCR:

- the security state in which the Vector Catch debug event can occur
- how the corresponding vector address depends on the configuration settings.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

これらのレジスタの詳細については、次のセクションを参照して下さい。

– *c12、ベクタ ベースアドレスレジスタ (VBAR)* P. B3-148

– *c12、モニタベクタ ベースアドレス レジスタ (MVBAR)* P. B3-149

注

リセット例外ベクタのアドレスがベクタ ベースアドレスの値に依存することはありません。また、SCTLR.VE = 1 の場合、IRQ および FIQ 例外ベクタのアドレスはベクタ ベースアドレスの値に依存しません。詳細については、P. C10-73 の表 C10-16 を参照して下さい。

- モニタモード ベクタでのセキュア状態のベクタキャッチを除いて、SCTLR の例外ベクタ構成について
  – SCTLR.V ビットが通常例外ベクタと上位例外ベクタのどちらにプログラムされているか
  – FIQ および IRQ 例外ベクタのキャッチの場合、SCTLR.VE ビットのプログラム

ベクタキャッチデバッグ イベントの生成は、プロセッサのセキュリティ状態にも依存します。

- 非セキュア状態ベクタキャッチは、非セキュア状態でのみ生成されます。
- セキュア状態ベクタキャッチは、セキュア状態でのみ生成されます。
- v6 デバッグと v6.1 デバッグでは、リセット ベクタキャッチはセキュア状態でのみ生成されます。

v7 デバッグでリセット ベクタキャッチが可能な場合、プロセッサのセキュリティ状態にかかわらずリセット ベクタキャッチが生成されます。

ベクタキャッチデバッグ イベントの生成では、SCR.NS を除くセキュア構成レジスタ (SCR) の値は考慮されません。例えば、DBGVCR のビット [14] が 1 にセットされて、モニタモード ベクタについてセキュア状態の IRQ をキャッチするようにプログラムされている場合、プロセッサがセキュア状態のとき、(MVBAR + `0x18`) からの命令プリフェッチすべてについて、ベクタキャッチデバッグ イベントが生成されます。このデバッグイベントは、SCR が IRQ を IRQ モードで処理するようにプログラムされている場合でも生成されます。

P. C10-71 の表 C10-15 は、DBGVCR のアクティブな各ビットについて、次の内容を示しています。

- ベクタキャッチデバッグ イベントが発生するセキュリティ状態
- 対応するベクタアドレスが、構成設定にどのように依存するか

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C10-16 Vector catch operation, when Security Extensions are implemented

DBGVCR bit	Vector catch enable		Security state	Configured exception vectors	
				Normal (V == 0)	High (V == 1)
Non-secure state vector catches					
[31]	FIQ	VE == 0	Non-secure	$VBAR_{NS}$ + 0x0000001C	0xFFFF001C
		VE == 1	Non-secure	Most recent Non-secure FIQ address[a]	
[30]	IRQ	VE == 0	Non-secure	$VBAR_{NS}$ + 0x00000018	0xFFFF0018
		VE == 1	Non-secure	Most recent Non-secure IRQ address[a]	
[28]	Data Abort		Non-secure	$VBAR_{NS}$ + 0x00000010	0xFFFF0010
[27]	Prefetch Abort		Non-secure	$VBAR_{NS}$ + 0x0000000C	0xFFFF000C
[26]	SVC		Non-secure	$VBAR_{NS}$ + 0x00000008	0xFFFF0008
[25]	Undefined Instruction		Non-secure	$VBAR_{NS}$ + 0x00000004	0xFFFF0004
Secure state vector catches on Monitor mode vectors					
[15]	FIQ		Secure	MVBAR + 0x0000001C	
[14]	IRQ		Secure	MVBAR + 0x00000018	
[12]	Data Abort		Secure	MVBAR + 0x00000010	
[11]	Prefetch Abort		Secure	MVBAR + 0x0000000C	
[10]	SMC		Secure	MVBAR + 0x00000008	

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C10-16 セキュリティ拡張機能が実装されている場合のベクタキャッチ操作

DBGVCR のビット	ベクタキャッチイネーブル		セキュリティ状態	構成されている例外ベクタ	
				通常 (V = 0)	上位 (V = 1)
非セキュア状態のベクタキャッチ					
[31]	FIQ	VE == 0	非セキュア	$VBAR_{NS}$ + 0x0000001C	0xFFFF001C
		VE == 1	非セキュア	最も新しい非セキュア FIQ アドレス[a]	
[30]	IRQ	VE == 0	非セキュア	$VBAR_{NS}$ + 0x00000018	0xFFFF0018
		VE == 1	非セキュア	最も新しい非セキュア IRQ アドレス[a]	
[28]	データアボート		非セキュア	$VBAR_{NS}$ + 0x00000010	0xFFFF0010
[27]	プリフェッチアボート		非セキュア	$VBAR_{NS}$ + 0x0000000C	0xFFFF000C
[26]	SVC		非セキュア	$VBAR_{NS}$ + 0x00000008	0xFFFF0008
[25]	未定義命令		非セキュア	$VBAR_{NS}$ + 0x00000004	0xFFFF0004
モニタモード ベクタに対するセキュア状態でのベクタキャッチ					
[15]	FIQ		セキュア	MVBAR + 0x0000001C	
[14]	IRQ		セキュア	MVBAR + 0x00000018	
[12]	データアボート		セキュア	MVBAR + 0x00000010	
[11]	プリフェッチアボート		セキュア	MVBAR + 0x0000000C	
[10]	SMC		セキュア	MVBAR + 0x00000008	

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C10-16 Vector catch operation, when Security Extensions are implemented (continued)

DBGVCR bit	Vector catch enable		Security state	Configured exception vectors	
				Normal (V == 0)	High (V == 1)
Secure state vector catches on exception mode vectors					
[7]	FIQ	VE = 0	Secure	$VBAR_S$ + 0x0000001C	0xFFFF001C
		VE = 1	Secure	Most recent Secure FIQ address[a]	
[6]	IRQ	VE = 0	Secure	$VBAR_S$ + 0x00000018	0xFFFF0018
		VE = 1	Secure	Most recent Secure IRQ address[a]	
[4]	Data Abort		Secure	$VBAR_S$ + 0x00000010	0xFFFF0010
[3]	Prefetch Abort		Secure	$VBAR_S$ + 0x0000000C	0xFFFF000C
[2]	SVC		Secure	$VBAR_S$ + 0x00000008	0xFFFF0008
[1]	Undefined Instruction		Secure	$VBAR_S$ + 0x00000004	0xFFFF0004
Reset vector catch[b]					
[0]	Reset		[b]	0x00000000	0xFFFF0000

a. For more information see *Vector catch debug events and vectored interrupt support* on page C3-22.
b. The value of the Reset vector is always independent of the Vector Base Address Register values. The security state dependence of Reset vector catches depends on the Debug architecture version. In v7 Debug, Reset vector catches are generated regardless of the security state of the processor. In v6 Debug and v6.1 Debug, Reset vector catches are generated only in Secure state.

In a v6.1 Debug implementation on a processor that implements the Security Extensions but does not implement DBGVCR bits [31, 30, 28:25, 15:14, 12:10]:

- in Non-secure state, bits [7:6, 4:1] apply to offsets from $VBAR_{NS}$.
- in Secure state, bits [7:6, 4:1] apply to offsets from $VBAR_S$ and bits [7:6, 4:2] also apply to offsets from MVBAR.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 C10-16 セキュリティ拡張機能が実装されている場合のベクタキャッチ操作（続き）

DBGVCR のビット	ベクタキャッチイネーブル		セキュリティ状態	構成されている例外ベクタ	
				通常 (V = 0)	上位 (V = 1)
例外モード ベクタに対するセキュア状態でのベクタキャッチ					
[7]	FIQ	VE = 0	セキュア	$VBAR_S$ + 0x0000001C	0xFFFF001C
		VE = 1	セキュア	最も新しいセキュア FIQ アドレス[a]	
[6]	IRQ	VE = 0	セキュア	$VBAR_S$ + 0x00000018	0xFFFF0018
		VE = 1	セキュア	最も新しいセキュア IRQ アドレス[a]	
[4]	データアボート		セキュア	$VBAR_S$ + 0x00000010	0xFFFF0010
[3]	プリフェッチアボート		セキュア	$VBAR_S$ + 0x0000000C	0xFFFF000C
[2]	SVC		セキュア	$VBAR_S$ + 0x00000008	0xFFFF0008
[1]	未定義命令		セキュア	$VBAR_S$ + 0x00000004	0xFFFF0004
リセット ベクタキャッチ[b]					
[0]	リセット		[b]	0x00000000	0xFFFF0000

a. 詳細については、*ベクタキャッチデバッグ イベントとベクタ割り込みのサポート* P. C3-22 を参照して下さい。
b. リセットベクタの値は、ベクタベースアドレス レジスタの値とは常に無関係です。リセット ベクタキャッチがセキュリティ状態に依存するかどうかは、デバッグアーキテクチャのバージョンによって異なります。v7 デバッグでは、プロセッサのセキュリティ状態にかかわらずリセット ベクタキャッチが生成されます。v6 デバッグと v6.1 デバッグでは、リセット ベクタキャッチはセキュア状態でのみ生成されます。

セキュリティ拡張機能を実装し、DBGVCR のビット [31、30、28:25、15:14、12:10] を実装していないプロセッサでの v6.1 デバッグ実装では、次の規則が適用されます。

- 非セキュア状態では、ビット [7:6、4:1] は $VBAR_{NS}$ からのオフセットに適用されます。
- セキュア状態では、ビット [7:6、4:1] は $VBAR_S$ からのオフセットに適用され、ビット [7:6、4:2] も MVBAR からのオフセットに適用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.6 OS Save and Restore registers, v7 Debug only

Support for the OS Save and Restore mechanism registers depends on the Debug architecture version:

v6 Debug and v6.1 Debug

These registers are not defined.

v7 Debug If an implementation supports debug over power-down, then it must implement the OS Save and Restore mechanism registers. On SinglePower systems, and on any other system that does not support debug over power-down, it is IMPLEMENTATION DEFINED whether the OS Save and Restore mechanism is implemented.

Any implementation that does not support the OS Save and Restore mechanism must implement the DBGOSLSR as RAZ.

This section describes the registers that provide the OS Save and Restore mechanism in the following subsections:

- *OS Lock Access Register (DBGOSLAR)*
- *OS Lock Status Register (DBGOSLSR)* on page C10-76
- *OS Save and Restore Register (DBGOSSRR)* on page C10-77.

In addition, the Event Catch Register enables the generation of a debug event when the OS Lock is unlocked. This register is described in *Event Catch Register (DBGECR)* on page C10-78.

C10.6.1 OS Lock Access Register (DBGOSLAR)

The OS Lock Access Register, DBGOSLAR, provides a lock for the debug registers. When the registers have been locked any access to the registers returns a slave-generated error response. Writing the key value to the DBGOSLAR has the side effect of resetting the internal counter for the OS Save or Restore operation.

You must use the DBGOSLSR to check the current status of the lock, see *OS Lock Status Register (DBGOSLSR)* on page C10-76.

The DBGOSLAR is:

- debug register 192, at offset `0x300`
- a write-only register
- only defined in v7 Debug
- when the Security Extensions are implemented, a Common register.

Note

In a v7 Debug implementation that does not implement the OS Save and Restore mechanism, register 192 ignores writes.

In v6 Debug and v6.1 Debug, register 192 is not defined.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.6 OS 保存および復元レジスタ（v7 デバッグのみ）

OS 保存および復元機構レジスタのサポートは、デバッグアーキテクチャのバージョンによって異なります。

v6 デバッグおよび v6.1 デバッグ

これらのレジスタは定義されていません。

v7 デバッグ 実装で電力オフをまたぐデバッグをサポートする場合、その実装には OS 保存および復元機構レジスタを実装する必要があります。SinglePower システム、および電力オフをまたぐデバッグをサポートしない他のシステムでは、OS 保存および復元機構を実装するかどうかは実装定義です。

OS 保存および復元機構をサポートしない実装では、DBGOSLSR を RAZ として実装する必要があります。

このセクションでは、OS 保存および復元機構を提供するレジスタについて、次のサブセクションで説明します。

- *OS ロックアクセス レジスタ (DBGOSLAR)*
- *OS ロックステータス レジスタ (DBGOSLSR)* P. C10- 76
- *OS 保存および復元レジスタ (DBGOSSRR)* P. C10- 77

また、イベントキャッチ レジスタは、OS ロックが解除されているときにデバッグイベントを生成するために使用されます。このレジスタについては、*イベントキャッチ レジスタ (DBGECR)* P. C10- 78 を参照して下さい。

C10.6.1 OS ロックアクセス レジスタ (DBGOSLAR)

OS ロックアクセス レジスタ (DBGOSLAR) は、デバッグレジスタのロックを提供します。レジスタがロックされている場合、そのレジスタへのアクセスはすべて、スレーブにより生成されるエラー応答を返します。DBGOSLAR にキー値を書き込むと、副作用として、OS 保存または復元操作の内部カウンタがリセットされます。

現在のロック状態をチェックするには、DBGOSLSR を使用する必要があります。*OS ロックステータス レジスタ (DBGOSLSR)* P. C10- 76 を参照して下さい。

DBGOSLAR の特徴は次のとおりです。

- デバッグレジスタ 192、オフセット `0x300`
- 書き込み専用レジスタ
- v7 デバッグでのみ定義されています。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

注

OS 保存および復元機構を実装していない v7 デバッグ実装では、レジスタ 192 への書き込みは無視されます。

v6 デバッグおよび v6.1 デバッグでは、レジスタ 192 は定義されていません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the DBGOSLAR is:

OS Lock Access, bits [31:0]

Writing the key value `0xC5ACCE55` to this field locks the debug registers, and resets the internal counter for the OS Save or Restore operation.

Writing any other value to this register unlocks the debug registers if they are locked.

For details of error responses when accessing the debug registers, see *Access permissions* on page C6-26.

If bit [0] of the Event Catch Register is set to 1 at the point when the OS Lock is unlocked, an OS Unlock Catch debug event is generated, see *Event Catch Register (DBGECR)* on page C10-78.

C10.6.2 OS Lock Status Register (DBGOSLSR)

The OS Lock Status Register, DBGOSLSR, provides status information for the OS Lock.

The DBGOSLSR is:

- debug register 193, at offset `0x304`
- a read-only register
- only defined in v7 Debug
- when the Security Extensions are implemented, a Common register.

Note

In any v7 Debug implementation you can read the DBGOSLSR to detect whether the OS Save and Restore mechanism is implemented. If it is not implemented the read of the DBGOSLSR returns zero.

In v6 Debug and v6.1 Debug, register 193 is not defined.

The format of the DBGOSLSR is:

Bits [31:3] Reserved, UNK.

32-bit access, bit [2]

This bit is always RAZ. It indicates that a 32-bit access is needed to write the key to the OS Lock Access Register.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DBGOSLAR のフォーマットは次のとおりです。

OS ロックアクセス、ビット [31:0]

このフィールドにキー値 `0xC5ACCE55` を書き込むと、デバッグレジスタがロックされ、OS 保存および復元操作の内部カウンタがリセットされます。

このレジスタに他の任意の値を書き込むと、デバッグレジスタがロックされていれば、ロック解除されます。

デバッグレジスタにアクセスしたときのエラー応答の詳細については、*アクセス許可* P. C6- 25 を参照して下さい。

OS ロックが解除された時点でイベントキャッチ レジスタのビット [0] が 1 にセットされている場合、OS ロック解除キャッチ デバッグイベントが生成されます。*イベントキャッチ レジスタ (DBGECR)* P. C10- 78 を参照して下さい。

C10.6.2 OS ロックステータス レジスタ (DBGOSLSR)

OS ロックステータス レジスタ (DBGOSLSR) は、OS ロックのステータス情報を提供します。

DBGOSLSR の特徴は次のとおりです。

- デバッグレジスタ 193、オフセット `0x304`
- 読み出し専用レジスタ
- v7 デバッグでのみ定義されています。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

注

任意の v7 デバッグ実装では、DBGOSLSR を読み出すと、OS 保存および復元機構が実装されているかどうかを検出できます。実装されていない場合、DBGOSLSR を読み出すと 0 が返されます。

v6 デバッグおよび v6.1 デバッグでは、レジスタ 193 は定義されていません。

DBGOSLSR のフォーマットは次のとおりです。

ビット [31:3] 予約、UNK

32 ビットアクセス、ビット [2]

このビットは常に RAZ です。OS ロックアクセス レジスタへキーを書き込むため、32 ビットアクセスが必要なことを示しています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Locked, bit [1]

This bit indicates the status of the OS Lock. The possible values are:

0 Lock not set.

1 Lock set. Writes to debug registers are ignored.

The OS Lock is set or cleared by writing to the DBGOSLAR, see *OS Lock Access Register (DBGOSLAR)* on page C10-75.

On a debug logic reset the state of the OS Lock and the value of this bit are IMPLEMENTATION DEFINED. If the implementation includes the recommended external debug interface they are determined by the value of the **DBGOSLOCKINIT** signal:

DBGOSLOCKINIT LOW

The lock is not set, and the Locked bit is 0

DBGOSLOCKINIT HIGH

The lock is set, and the Locked bit is 1.

Lock implemented, bit [0]

This bit reads 1 if it is possible to set the OS Lock for this processor.

If this bit reads 0, OS Lock and the OS Save and Restore mechanism are not implemented and the entire register is RAZ.

C10.6.3 OS Save and Restore Register (DBGOSSRR)

The OS Save and Restore Register, DBGOSSRR, enables the entire debug logic state of the processor to be either saved or restored, by performing a series of reads or writes of the DBGOSSRR. The register works in conjunction with an internal sequence counter to perform the OS Save or Restore operation.

The DBGOSSRR is:

- debug register 194, at offset `0x308`
- a read/write register
- only defined in v7 Debug
- when the Security Extensions are implemented, a Common register.

Note

- In a v7 Debug implementation that does not implement the OS Save and Restore mechanism, register 194 is RAZ/WI.
- For more information about access permissions in an implementation that includes the OS Save and Restore mechanism but does not provide access to the DBGOSSRR through the external debug interface, see the Note in *The OS Save and Restore mechanism* on page C6-8.

In v6 Debug and v6.1 Debug, register 194 is not defined.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ロック、ビット [1]

このビットは、OS ロックのステータスを示します。可能な値は次のとおりです。

0 ロックが設定されていません。

1 ロックが設定されています。デバッグレジスタへの書き込みは無視されます。

OS ロックは、DBGOSLAR への書き込みにより設定またはクリアされます。*OS ロックアクセス レジスタ (DBGOSLAR)* P. C10-75 を参照して下さい。

デバッグロジック リセット時の OS ロック状態、およびこのビットの値は実装定義です。推奨される外部デバッグインタフェースが実装に含まれる場合、これらは DBGOSLOCKINIT 信号の値によって決定されます。

DBGOSLOCKINIT が LOW の場合

ロックは設定されず、ロックビットは 0 です。

DBGOSLOCKINIT が HIGH の場合

ロックは設定され、ロックビットは 1 です。

ロック実装、ビット [0]

このプロセッサに OS ロックを設定可能な場合、このビットは 1 として読み出されます。

このビットが 0 として読み出される場合、OS ロックと、OS 保存および復元機構は実装されておらず、レジスタ全体が RAZ です。

C10.6.3 OS 保存および復元レジスタ (DBGOSSRR)

OS 保存および復元レジスタ (DBGOSSRR) は、このレジスタに対する一連の読み出しまたは書き込みを実行し、プロセッサのデバッグロジック状態の全体を保存または復元するために使用されます。このレジスタは、OS 保存または復元操作を実行するため、内部のシーケンスカウンタと組み合わせて動作します。

DBGOSSRR の特徴は次のとおりです。

- デバッグレジスタ 194、オフセット `0x308`
- 読み出し / 書き込みレジスタ
- v7 デバッグでのみ定義されています。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

注

- OS 保存および復元機構を実装していない v7 デバッグ実装では、レジスタ 194 は RAZ/WI です。
- OS 保存および復元機構を含み、外部デバッグインタフェース経由の DBGOSSRR へのアクセスを提供しない実装におけるアクセス許可の詳細については、*OS 保存および復元機構* P. C6-8 の注を参照して下さい。

v6 デバッグおよび v6.1 デバッグでは、レジスタ 194 は定義されていません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the DBGOSSRR is:

31 … 0
OS Save or Restore value

OS Save or Restore value, bits [31:0]

After a write to the DBGOSLAR to lock the debug registers, the first access to the DBGOSSRR must be a read:

- when performing an OS Save sequence this read returns the number of reads from to the DBGOSSRR that are needed to save the entire debug logic state
- when performing an OS Restore sequence the value of this read is UNKNOWN and must be discarded.

After that first read access:

- a read of this register returns the next debug logic state value to be saved
- a write to this register restores the next debug logic state value.

Before accessing the DBGOSSRR, you must write to the DBGOSLAR to set the OS Lock, see *OS Lock Access Register (DBGOSLAR)* on page C10-75. This write to the DBGOSLAR resets the internal counter for the OS Save or Restore operation.

The result is UNPREDICTABLE if:

- you access the DBGOSSRR when the OS Lock is not set
- after setting the OS Lock, the first access to the DBGOSSRR is not a read.

See *The OS Save and Restore mechanism* on page C6-8 for a description of using the OS Save and Restore mechanism registers.

C10.6.4 Event Catch Register (DBGECR)

The Event Catch Register, DBGECR, configures the debug logic to generate a debug event when the OS Lock is unlocked.

The DBGECR is:

- debug register 9, at offset `0x024`
- a read/write register
- only defined in v7 Debug
- when the Security Extensions are implemented, a Common register.

In v6 Debug and v6.1 Debug, register 9 is not defined.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DBGOSSRR のフォーマットは次のとおりです。

31 … 0
OS 保存または復元の値

OS 保存または復元の値、ビット [31:0]

DBGOSLAR への書き込みによりデバッグレジスタをロックした後で、DBGOSSRR への最初のアクセスは読み出しの必要があります。

- OS 保存シーケンスを実行しているとき、この読み出しでは、デバッグロジック状態の全体を保存するために必要な、DBGOSSRR からの読み出し回数が返されます。
- OS 復元シーケンスを実行しているとき、この読み出しで返される値は未知で、値を破棄する必要があります。

最初の読み出し以後のアクセスでは、次の動作が行われます。

- このレジスタの読み出しでは、保存される次のデバッグロジック状態の値が返されます。
- このレジスタへの書き込みでは、次のデバッグロジック状態の値が復元されます。

DBGOSSRR にアクセスする前に、DBGOSLAR への書き込みによって OS ロックを設定する必要があります。*OS ロックアクセス レジスタ (DBGOSLAR)* P. C10-75 を参照して下さい。
DBGOSLAR への書き込みは、OS 保存または復元操作の内部カウンタをリセットします。

次の動作を行った場合、結果は予測不能です。

- OS ロックが設定されていないときに DBGOSSRR へアクセスした。
- OS ロックの設定後、DBGOSSRR への最初のアクセスが読み出しではなかった。

OS 保存および復元機構のレジスタの使用方法の説明については、*OS 保存および復元機構* P. C6-8 を参照して下さい。

C10.6.4 イベントキャッチ レジスタ (DBGECR)

イベントキャッチ レジスタ (DBGECR) は、OS ロックが解除されたときにデバッグイベントを生成するようにデバッグロジックを構成します。

DBGECR の特徴は次のとおりです。

- デバッグレジスタ 9、オフセット `0x024`
- 読み出し / 書き込みレジスタ
- v7 デバッグでのみ定義されている。
- セキュリティ拡張機能が実装されている場合、共通レジスタである。

v6 デバッグおよび v6.1 デバッグでは、レジスタ 9 は定義されていません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the DBGECR is:

Bits [31:1] Reserved. UNK/SBZP.

OS Unlock Catch, bit [0]

When this bit is set to 1, an OS Unlock Catch debug event is generated when the OS Lock is unlocked by writing to the DBGOSLAR, see *OS Lock Access Register (DBGOSLAR)* on page C10-75. The possible values of this bit are:

0 OS Unlock Catch disabled

1 OS Unlock Catch enabled.

The debug logic reset value of this bit is 0.

The OS Unlock Catch bit is part of the OS Save and Restore mechanism. If an implementation supports debug over power-down it must support the OS Save and Restore mechanism, including the OS Unlock Catch debug event. If an implementation does not support debug over power-down, it is IMPLEMENTATION DEFINED whether the OS Save and Restore mechanism and the OS Unlock Catch debug event are supported. If the OS Unlock Catch debug event is not supported then this bit is RAZ/WI.

The OS Unlock Catch debug event is a Halting debug event, see *Halting debug events* on page C3-38. If a debugger is monitoring an application running on top of an OS with OS Save and Restore capability, this event indicates the right time for the debug session to continue.

Note

The OS Unlock Catch debug event is generated only on clearing of the OS Lock, that is, on the transition of OS Lock from locked to unlocked.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DBGECR のフォーマットは次のとおりです。

ビット [31:1] 予約。UNK/SBZP

OS ロック解除キャッチ、ビット [0]

このビットが 1 にセットされているとき、DBGOSLAR への書き込みにより OS ロックが解除されると、OS ロック解除キャッチ デバッグイベントが生成されます。*OS ロックアクセス レジスタ (DBGOSLAR)* P. C10- 75 を参照して下さい。このビットの可能な値は次のとおりです。

0 OS ロック解除キャッチは不可能です。

1 OS ロック解除キャッチが可能です。

デバッグロジック リセット時のこのビットの値は 0 です。

OS ロック解除キャッチビットは、OS 保存および復元機構の一部です。実装で、電力オフをまたぐデバッグをサポートする場合、OS ロック解除キャッチ デバッグイベントを含む OS 保存および復元機構をサポートする必要があります。実装で、電力オフをまたぐデバッグをサポートしない場合、OS 保存および復元機構と OS ロック解除キャッチ デバッグイベントがサポートされているかどうかは実装定義です。OS ロック解除キャッチ デバッグイベントがサポートされていない場合、このビットは RAZ/WI です。

OS ロック解除キャッチ デバッグイベントはホールトデバッグ イベントです。*ホールトデバッグイベント* P. C3- 38 を参照して下さい。デバッガが、OS 保存および復元機構を持つ OS の上で実行されているアプリケーションを監視している場合、このイベントは、デバッグセッションを継続すべき適切な時点を示します。

注

OS ロック解除キャッチ デバッグイベントは、OS ロックがクリアされたとき、つまり OS ロックがロック状態からロック解除状態に移行するときにのみ生成されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.7 Memory system control registers

Support for the Memory system control registers can depend on the Debug architecture version:

v6 Debug and v6.1 Debug

In some v6 Debug and v6.1 Debug implementations a Cache Behavior Override Register (CBOR) is provided in an IMPLEMENTATION DEFINED region of the CP15 register space. In addition, particularly in v6.1 Debug implementations, the Debug State MMU Control Register (DBGDSMCR) and Debug State Cache Control Register (DBGDSCCR) might be implemented as IMPLEMENTATION DEFINED extensions to CP14, as described below.

v6 Debug and v6.1 Debug do not require these registers. However, ARM recommends these features to assist debuggers to maintaining memory coherency, avoiding costly explicit coherency operations.

v7 Debug In v7 Debug, the DBGDSMCR and DBGDSCCR are required, but there can be IMPLEMENTATION DEFINED limits on their behavior. The CP15 register CBOR remains IMPLEMENTATION DEFINED.

The Memory system control registers are described in the subsections:

- *Debug State Cache Control Register (DBGDSCCR)* on page C10-81
- *Debug State MMU Control Register (DBGDSMCR)* on page C10-84.

The Debug State Cache Control Register (DBGDSCCR) and Debug State MMU Control Register (DBGDSMCR) control cache and TLB behavior for memory operations issued by a debugger when the processor is in Debug state. They enable a debugger to request the minimum amount of intrusion to the processor caches, as permitted by the implementation. It is IMPLEMENTATION DEFINED what levels of cache and TLB are controlled by these requests, and it is IMPLEMENTATION DEFINED to what extent the intrusion is limited.

The DBGDSCCR also provides a mechanism for a debugger to force writes to memory through to the point of coherency without the overhead of issuing additional operations.

The DBGDSCCR and DBGDSMCR controls must apply for all memory operations issued in Debug state when DBGDSCR.ADAdiscard, the Asynchronous Data Aborts Discarded bit, is set to 1. It is IMPLEMENTATION DEFINED whether memory operations issued in Debug state whilst this bit is not set to 1 are affected by the DBGDSCCR and DBGDSMCR.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C10.7 メモリシステム制御レジスタ

メモリシステム制御レジスタのサポートは、デバッグアーキテクチャのバージョンによって異なります。

v6 デバッグと v6.1 デバッグ

v6 デバッグと v6.1 デバッグの一部の実装では、CP15 レジスタ空間の実装定義領域に、キャッシュ動作オーバライドレジスタ (CBOR) が提供されています。さらに、v6.1 デバッグ実装では特に、デバック状態 MMU 制御レジスタ (DBGDSMCR) とデバッグ状態キャッシュ制御レジスタ (DBGDSCCR) は、以下に説明されているように、CP14 の実装定義拡張機能として実装されていることがあります。

v6 デバッグと v6.1 デバッグでは、これらのレジスタは必須ではありません。ただし、デバッガがメモリコヒーレンシを維持し、コストの高い明示的なコヒーレンシ操作を避けるために役立つため、これらの機能を実装することを推奨します。

v7 デバッグ v7 デバッグでは DBGDSMCR および DBGDSCCR が必須ですが、これらのレジスタの動作に実装定義の制限を設けることができます。CP15 レジスタ CBOR は実装定義のままです。

メモリシステム制御レジスタについては、次のサブセクションで説明されています。

- *デバッグ状態キャッシュ制御レジスタ (DBGDSCCR)* P. C10- 81
- *デバッグ状態 MMU 制御レジスタ (DBGDSMCR)* P. C10- 84

デバッグ状態キャッシュ制御レジスタ (DBGDSCCR) およびデバッグ状態 MMU 制御レジスタ (DBGDSMCR) は、プロセッサがデバッグ状態のときにデバッガにより発行されるメモリ操作について、キャッシュと TLB の動作を制御します。デバッガはこれらのレジスタにより、実装によって許可される限り最小の、プロセッサキャッシュへの侵襲性を要求できます。これらの要求により、どのレベルのキャッシュと TLB が制御されるかは実装定義です。また、侵襲性がどれだけの範囲に制限されるかも実装定義です。

DBGDSCCR は、デバッガが、追加操作を発行するオーバヘッドなしに、コヒーレンシ時点までメモリへの書き込みを強制するための機構も提供します。

DBGDSCCR および DBGDSMCR の制御は、DBGDSCR.ADAdiscard（非同期データアボート破棄ビット）が 1 にセットされているときにデバッグ状態で発行された、すべてのメモリ操作に適用される必要があります。このビットが 1 にセットされていないときにデバッグ状態で発行されたメモリ操作が、DBGDSCCR および DBGDSMCR の影響を受けるかどうかは、実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.7.1 Debug State Cache Control Register (DBGDSCCR)

The Debug State Cache Control Register, DBGDSCCR, controls cache behavior when the processor is in Debug state.

The DBGDSCCR is:

- debug register 10, at offset `0x028`
- a read/write register, with some bits that might not be implemented and therefore are RAZ/WI
- required in v7 Debug
- when the Security Extensions are implemented, a Common register.

It is IMPLEMENTATION DEFINED whether the DBGDSCCR is included in a v6 Debug or v6.1 Debug implementation.

The format of the DBGDSCCR is:

Bits [31:3] Reserved, UNK/SBZP.

Force Write-Through, nWT, bit [2]

The possible values of this bit are:

0 Force Write-Through behavior for memory operations issued by a debugger when the processor is in Debug state

1 Normal operation for memory operations issued by a debugger when the processor is in Debug state.

In Debug state, if the nWT bit is set to 0, when a write to memory completes the effect of the write must be visible at all levels of memory to the point of coherency. This means a debugger can write through to the point of coherency without having to perform any cache clean operations.

If implemented, the nWT control must act at all levels of memory to the point of coherency.

If the nWT control is not implemented this bit is RAZ/WI.

Note

nWT does not force the ordering of writes, and does not force writes to complete immediately. A debugger might have to insert a barrier operations to ensure ordering.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.7.1 デバッグ状態キャッシュ制御レジスタ (DBGDSCCR)

デバッグ状態キャッシュ制御レジスタ (DBGDSCCR) は、プロセッサがデバッグ状態のときにキャッシュの動作を制御します。

DBGDSCCR の特徴は次のとおりです。

- デバッグレジスタ 10、オフセット `0x028`
- 読み出し / 書き込みレジスタで、一部のビットは実装されておらず RAZ/WI の可能性がある。
- v7 デバッグでは必須である。
- セキュリティ拡張機能が実装されている場合、共通レジスタである。

v6 デバッグまたは v6.1 デバッグの実装に DBGDSCCR が含まれるかどうかは実装定義です。

DBGDSCCR のフォーマットは次のとおりです。

ビット [31:3] 予約、UNK/SBZP

強制ライトスルー、nWT、ビット [2]

このビットの可能な値は次のとおりです。

0 プロセッサがデバッグ状態のときにデバッガにより発行されたメモリ操作について、強制的にライトスルー動作を行います。

1 プロセッサがデバッグ状態のときにデバッガにより発行されたメモリ操作について、通常の動作を行います。

デバック状態で nWT ビットが 0 にセットされている場合、メモリへの書き込みが完了する時点で、その書き込みの影響は、コヒーレンシ時点までのメモリのすべてのレベルについて、可視の必要があります。これは、デバッガはキャッシュのクリーニング操作を行う必要なく、コヒーレンシ時点までのライトスルーを実行できることを意味します。

実装されている場合、nWT の制御は、メモリのすべてのレベルについて、コヒーレンシ時点まで動作する必要があります。

nWT 制御が実装されていない場合、このビットは RAZ/WI です。

注

nWT は書き込みの順序を強制せず、書き込みが直ちに完了することも強制しません。デバッガは、順序付けを保証するためにバリア操作を挿入することが必要な場合があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Cache linefill and eviction bits, bits [1:0]

Either or both of these bits might not be implemented, in which case the bit is RAZ/WI. If implemented these bits are:

nIL, bit [1] Instruction cache, where separate data and instruction caches are implemented.

nDL, bit [0] Data or unified cache.

The possible values of an implemented bit are:

0 Request disabling of cache linefills and evictions for memory operations issued by a debugger when the processor is in Debug state

1 Normal operation of cache linefills and evictions for memory operations issued by a debugger when the processor is in Debug state.

When cache linefill and eviction is disabled, all memory accesses that would be checked against a cache are checked against the cache. If a match is found, the cached result is used. If no match is found the next level of memory is used, but the result is not cached, and no cache entries are evicted.

The *next level of memory* can refer to looking in the next level of cache, or to accessing external memory, depending on the numbers of levels of cache implemented.

When the processor is in Debug state, cache maintenance operations are not affected by the nDL and nIL control bits, and have their normal architecturally-defined behavior.

The memory hint instructions `PLD` and `PLI` have UNPREDICTABLE behavior in Debug state when the corresponding nDL or nIL control bit is set to 1.

The debug logic reset value of the DBGDSCCR depends on the ARM Debug architecture version:

v7 Debug Debug logic reset values are UNKNOWN. Before issuing operations through the DBGITR with the processor in Debug state, a debugger must ensure that the DBGDSCCR has a defined state.

ARMv6 All defined bits reset to 0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

キャッシュラインフィルおよび退出ビット、ビット [1:0]

これらのビットの片方または両方が実装されていないことがあります。その場合、そのビットは RAZ/WI です。実装されている場合、これらのビットは次のとおりです。

nIL、ビット [1] データキャッシュと命令キャッシュが別に実装されている場合の命令キャッシュ

nDL、ビット [0] データキャッシュまたは統一キャッシュ

実装されている場合、これらのビットの可能な値は次のとおりです。

0 プロセッサがデバッグ状態のとき、デバッガにより発行されたメモリ操作について、キャッシュのラインフィルと退出の非稼動を要求します。

1 プロセッサがデバッグ状態のとき、デバッガにより発行されたメモリ操作について、キャッシュのラインフィルと退出は通常に動作します。

キャッシュのラインフィルと退出が非稼動のとき、キャッシュに対してチェックされるべきすべてのメモリアクセスは、そのキャッシュに対してチェックされます。一致が見つかった場合、キャッシュされた結果が使用されます。一致が見つからない場合、次のレベルのメモリが使用されますが、その結果はキャッシュされず、キャッシュのエントリの退出は行われません。

*次のレベルのメモリ*とは、実装されているキャッシュのレベル数により、キャッシュの次のレベルを探すこと、または外部メモリにアクセスすることを指します。

プロセッサがデバッグ状態のとき、キャッシュ保守操作は nDL および nIL 制御ビットの影響を受けず、アーキテクチャで定義された通常の動作を行います。

メモリヒント命令 `PLD` および `PLI` は、対応する nDL または nIL 制御ビットが 1 にセットされている場合、デバッグ状態での動作は予測不能です。

デバッグロジック リセット時の DBGDSCCR の値は、ARM デバッグアーキテクチャのバージョンによって異なります。

v7 デバッグ デバッグロジック リセット時の値は未知です。デバッガは、プロセッサがデバッグ状態のときに DBGITR 経由で操作を発行する前に、DBGDSCCR が定義済みの状態であることを確認する必要があります。

ARMv6 定義済みのビットはすべて 0 にリセットされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Permitted IMPLEMENTATION DEFINED limits

The DBGDSCCR is required. However, there can be IMPLEMENTATION DEFINED limits on its behavior. Table C10-17 lists some examples of possible options for implementations.

Table C10-17 Permitted IMPLEMENTATION DEFINED limits on DBGDSCCR behavior

Limit	Description	Notes
Full DBGDSCCR	Bits [2:0] implemented	-
No Write-Back support	Bit [2] is RAZ/WI	-
No Write-Through support	Bit [2] is RAZ/WI	Force Write-Through feature not supported. If Secure User halting debug is supported the implementation must provide cache clean operations in Debug state, see *Access to specific cache management functions in Debug state* on page C5-25.
No I-cache control	Bit [1] is RAZ/WI	Instruction cache linefill and eviction disable features not implemented. Instruction fetches are disabled in Debug state. For most implementations no instruction cache accesses take place in Debug state, and nIL is not required.
Unified cache	Bit [1] is RAZ/WI	-
Cache evictions always enabled	-	nIL and nDL disable cache linefills in Debug state. However cache evictions might still take place even when these control bits are set to 0.
No linefill control	Bits [1:0] are RAZ/WI	No cache linefill and eviction disable features are implemented.

Interaction with Cache Behavior Override Register

An IMPLEMENTATION DEFINED Cache Behavior Override Register (CBOR) might also be implemented in CP15.

Table C10-18 on page C10-84 shows, for a processor that implements both the Debug state Cache Control Register (DBGDSCCR) and the CBOR, the relative precedence of the CBOR and the DBGDSCCR according to the state of the processor.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

許容される実装定義の制限

DBGDSCCR は必須です。ただし、その動作には実装定義の制限を設けることができます。実装で可能なオプションについて、いくつかの例を表 C10-17 に示します。

表 C10-17 DBGDSCCR の動作について許容される実装定義の制限

制限	説明	注
完全な DBGDSCCR	ビット [2:0] が実装されます。	–
ライトバックのサポートなし	ビット [2] は RAZ/WI です。	–
ライトスルーのサポートなし	ビット [2] は RAZ/WI です。	ライトスルーの強制機能がサポートされていません。 セキュアユーザ ホールトデバッグがサポートされている場合、この実装ではデバッグ状態でのキャッシュのクリーニング操作を提供する必要があります。*デバッグ状態での特定のキャッシュ管理機能へのアクセス* P. C5-25 を参照して下さい。
I キャッシュ制御なし	ビット [1] は RAZ/WI です。	命令キャッシュのラインフィルおよび退出の非稼動化機能は実装されていません。 デバッグ状態での命令フェッチは不可能になります。ほとんどの実装では、デバック状態で命令キャッシュへのアクセスは行われず、nIL は必要ありません。
統一キャッシュ	ビット [1] は RAZ/WI です。	–
キャッシュの退出が常に可能	–	デバッグ状態で、nIL および nDL によりキャッシュのラインフィルが不可能になります。ただし、これらの制御ビットが 0 にセットされている場合も、キャッシュの退出は依然として行われる可能性があります。
ラインフィル制御なし	ビット [1:0] は RAZ/WI です。	キャッシュのラインフィルおよび退出の非稼動化機能は実装されていません。

キャッシュ動作オーバライドレジスタとの相互動作

CP15 には、実装定義のキャッシュ動作オーバライドレジスタ (CBOR) も実装されていることがあります。

デバッグ状態キャッシュ制御レジスタ (DBGDSCCR) と CBOR の両方を実装しているプロセッサについて、プロセッサの状態に応じた CBOR と DBGDSCCR の相対的な優先度を、P. C10-84 の表 C10-18 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C10-18 Interaction of CP15 Cache Behavior Override Register (CBOR) and DBGDSCCR

DBGDSCCR setting	CBOR setting	Debug state	Behavior
nWT = 1	WT = 0	X	Areas marked WB are Write-Back
nWT = X	WT = 0	No	Areas marked WB are Write-Back
nWT = X	WT = 1	X	Areas marked WB are Write-Through
nWT = 0	WT = X	Yes	Areas marked WB are Write-Through
nDL = 1	DL = 0	X	Data or unified cache linefills are enabled
nDL = X	DL = 0	No	Data or unified cache linefills are enabled
nDL = X	DL = 1	X	Data or unified cache linefills are disabled
nDL = 0	DL = X	Yes	Data or unified cache linefills are disabled
nIL = 1	IL = 0	X	Instruction cache linefills are enabled
nIL = X	IL = 0	No	Instruction cache linefills are enabled
nIL = X	IL = 1	X	Instruction cache linefills are disabled
nIL = 0	IL = X	Yes	Instruction cache linefills are disabled

A processor that does not implement Security Extensions has only WT, IL and DL settings in the CP15 Cache Behavior Override Register. Processors that implement Security Extensions can have separate settings for, for example, NS_WT and S_WT in the CP15 Cache Behavior Override Register. For brevity Table C10-18 does not show the full matrix of possibilities in this case. For the behavior on such a processor, duplicate Table C10-18:

- once for the Non-secure case, for example NS_WT
- once for the Secure case, for example S_WT.

C10.7.2 Debug State MMU Control Register (DBGDSMCR)

The Debug State MMU Control Register, DBGDSMCR, controls TLB behavior when the processor is in Debug state.

The DBGDSMCR is:

- debug register 11, at offset `0x02C`
- a read/write register, with some bits that might not be implemented and therefore are RAZ/WI
- required in v7 Debug
- when the Security Extensions are implemented, a Common register.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 C10-18 CP15 キャッシュ動作オーバライドレジスタ (CBOR) と DBGDSCCR との相互動作

DBGDSCCR の設定	CBOR の設定	デバッグ状態	動作
nWT = 1	WT = 0	X	WB にマークされている領域はライトバックです。
nWT = X	WT = 0	いいえ	WB にマークされている領域はライトバックです。
nWT = X	WT = 1	X	WB にマークされている領域はライトスルーです。
nWT = 0	WT = X	はい	WB にマークされている領域はライトスルーです。
nDL = 1	DL = 0	X	データまたは統一キャッシュのラインフィルが可能です。
nDL = X	DL = 0	いいえ	データまたは統一キャッシュのラインフィルが可能です。
nDL = X	DL = 1	X	データまたは統一キャッシュのラインフィルが不可能です。
nDL = 0	DL = X	はい	データまたは統一キャッシュのラインフィルが不可能です。
nIL = 1	IL = 0	X	命令キャッシュのラインフィルが可能です。
nIL = X	IL = 0	いいえ	命令キャッシュのラインフィルが可能です。
nIL = X	IL = 1	X	命令キャッシュのラインフィルが不可能です。
nIL = 0	IL = X	はい	命令キャッシュのラインフィルが不可能です。

セキュリティ拡張機能が実装されていないプロセッサには、CP15 キャッシュ動作オーバライドレジスタの WT、IL、DL 設定のみが存在します。セキュリティ拡張機能が実装されているプロセッサには、CP15 キャッシュ動作オーバライドレジスタについて、NS_WT および S_WT など、独立した設定が存在することがあります。簡素化のため、このような場合に可能なすべてのマトリックスは、表 C10-18 には示されていません。このようなプロセッサの動作については、表 C10-18 を次のように重複させて下さい。

- 非セキュア状態、例えば NS_WT について 1 回
- セキュア状態、例えば S_WT について 1 回

C10.7.2 デバッグ状態 MMU 制御レジスタ (DBGDSMCR)

デバッグ状態 MMU 制御レジスタ (DBGDSMCR) は、プロセッサがデバッグ状態のときに TLB の動作を制御します。

DBGDSMCR の特徴は次のとおりです。

- デバッグレジスタ 11、オフセット `0x02C`
- 読み出し / 書き込みレジスタで、一部のビットは実装されておらず RAZ/WI の可能性がある。
- v7 デバッグでは必須である。
- セキュリティ拡張機能が実装されている場合、共通レジスタである。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

It is IMPLEMENTATION DEFINED whether the DBGDSMCR is included in a v6 Debug or v6.1 Debug implementation.

The format of the DBGDSMCR is:

Bits [31:4] Reserved, UNK/SBZP.

TLB matching bits, bits [3:2]

Either or both of these bits might not be implemented, in which case the bit is RAZ/WI. If implemented these bits are:

nIUM, bit [3] Instruction TLB matching bit, where separate Data and Instruction TLBs are implemented.

nDUM, bit [2] Data or Unified TLB matching bit.

The possible values of an implemented TLB matching bit are:

0 Request disabling of TLB matching for memory operations issued by a debugger when the processor is in Debug state

1 Normal operation of TLB matching for memory operations issued by a debugger when the processor is in Debug state.

When TLB matching is disabled, all memory accesses normally checked against a TLB are not checked against the TLB. For every access the next level of translation is performed. The results are not cached in the TLB, and no TLB entries are evicted. The next level of translation is used for every access.

The *next level of translation* might mean looking in the next level TLB, or doing a translation table walk, depending on the numbers of levels of TLB implemented.

Note

If TLB matching is disabled, and TLB maintenance functions have not been correctly performed by the system being debugged, for example, if the TLB has not been flushed following a change to the translation tables, memory accesses made by the debugger might not undergo the same virtual to physical memory mappings as the application being debugged.

A debugger can create temporary alternative memory mappings by altering the contents of the external translation tables and disabling all levels of TLB matching. However, for normal debugging operations, ARM recommends that the TLB Matching bit is set to 1.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

v6 デバッグまたは v6.1 デバッグの実装に DBGDSMCR が含まれるかどうかは実装定義です。

DBGDSMCR のフォーマットは次のとおりです。

ビット [31:4] 予約、UNK/SBZP

TLB 一致ビット、ビット [3:2]

これらのビットの片方または両方が実装されていないことがあります。その場合、そのビットは RAZ/WI です。実装されている場合、これらのビットは次のとおりです。

nIUM、ビット [3] データ TLB と命令 TLB が独立して実装されている場合の、命令 TLB 一致ビット。

nDUM、ビット [2] データまたは統一 TLB 一致ビット。

実装されている場合、TLB 一致ビットの可能な値は次のとおりです。

0 プロセッサがデバッグ状態のとき、デバッガにより発行されたメモリ操作について、TLB 一致の非稼動を要求します。

1 プロセッサがデバッグ状態のとき、デバッガにより発行されたメモリ操作について、通常の TLB 一致動作を行います。

TLB 一致が非稼動のとき、通常は TLB に対してチェックされるすべてのメモリアクセスは、TLB に対してチェックされません。すべてのアクセスについて、次のレベルの変換が実行されます。結果は TLB にキャッシュされず、TLB エントリの退出は行われません。すべてのアクセスについて、次のレベルの変換が使用されます。

*次のレベルの変換*とは、実装されている TLB のレベル数によって、次のレベルの TLB を探すこと、または変換テーブルウォークの実行を意味します。

注

TLB 一致が非稼動であり、デバッグ対象のシステムで TLB 保守機能が正しく実行されていない、例えば変換テーブルを変更した後で TLB がフラッシュされていない場合、デバッガにより行われるメモリアクセスは、デバッグ対象のアプリケーションと同一の仮想メモリから物理メモリへのマッピングで行われないことがあります。

デバッガは、外部変換テーブルの内容を変更し、すべてのレベルの TLB 一致を非稼動にして、代替のメモリマッピングを一時的に作成することができます。ただし、通常のデバッグ動作では、TLB 一致ビットを 1 にセットしておくことをお勧めします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

TLB loading bits, bits [1:0]

Either or both of these bits might not be implemented, in which case the bit is RAZ/WI. If implemented these bits are:

nIUL, bit [1] Instruction TLB loading bit, where separate Data and Instruction TLBs are implemented.

nDUL, bit [0] Data or Unified TLB loading bit.

The possible values of an implemented TLB loading bit are:

0 Request disabling of TLB load and flush for memory operations issued by a debugger when the processor is in Debug state

1 Normal operation of TLB loading and flushing for memory operations issued by a debugger when the processor is in Debug state.

When TLB load and flush is disabled, all memory accesses normally checked against a TLB are checked against the TLB. If a match is found, the cached result is used. If no match is found the next level of translation is performed, but the result is not cached in the TLB, and no TLB entries are evicted.

The *next level of translation* might mean looking in the next level TLB, or doing a translation table walk, depending on the numbers of levels of TLB implemented.

In Debug state, TLB maintenance operations are not affected by the nDUL and nIUL control bits, and have their normal architecturally-defined behavior.

The debug logic reset value of the DBGDSMCR depends on the ARM Debug architecture version:

v7 Debug Debug logic reset values are UNKNOWN. Before issuing operations through the DBGITR with the processor in Debug state, a debugger must ensure that the DBGDSMCR has a defined state.

v6 Debug, v6.1 Debug

All defined bits reset to 0.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

TLB ロードビット、ビット [1:0]

これらのビットの片方または両方が実装されていないことがあります。その場合、そのビットは RAZ/WI です。実装されている場合、これらのビットは次のとおりです。

nIUL、ビット [1] データ TLB と命令 TLB が独立して実装されている場合の、命令 TLB ロードビット。

nDUL、ビット [0] データまたは統一 TLB ロードビット。

実装されている場合、TLB ロードビットの可能な値は次のとおりです。

0 プロセッサがデバッグ状態のとき、デバッガにより発行されたメモリ操作について、TLB ロードおよびフラッシュの非稼動を要求します。

1 プロセッサがデバッグ状態のとき、デバッガにより発行されたメモリ操作について、通常の TLB ロードおよびフラッシュ動作を行います。

TLB ロードおよびフラッシュが非稼動のとき、通常は TLB に対してチェックされるすべてのメモリアクセスは、TLB に対してチェックされます。一致が見つかった場合、キャッシュされた結果が使用されます。一致が見つからない場合、次のレベルの変換が実行されますが、その結果は TLB にキャッシュされず、TLB エントリの退出は行われません。

*次のレベルの変換*とは、実装されている TLB のレベル数によって、次のレベルの TLB を探すこと、または変換テーブルウォークの実行を意味します。

デバッグ状態で、TLB 保守操作は nDUL および nIUL 制御ビットの影響を受けず、アーキテクチャで定義された通常の動作を行います。

デバッグロジック リセット時の DBGDSMCR の値は、ARM デバッグアーキテクチャのバージョンによって異なります。

v7 デバッグ デバッグロジック リセット時の値は未知です。デバッガは、プロセッサがデバッグ状態のときに DBGITR 経由で操作を発行する前に、DBGDSMCR が定義済みの状態であることを確認する必要があります。

v6 デバッグ、v6.1 デバッグ

定義済みのビットはすべて 0 にリセットされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Permitted IMPLEMENTATION DEFINED limits

The DBGDSMCR is required. However, there can be IMPLEMENTATION DEFINED limits on its behavior. Table C10-19 lists six permitted options for implementations. Some of these options are orthogonal.

Table C10-19 Permitted IMPLEMENTATION DEFINED limits on DBGDSCCR behavior

Limit	Description	Notes
Full DBGDSMCR	Bits [3:0] implemented	-
No I-TLB controls	Bits [3,1] are RAZ/WI	Instruction cache linefill and eviction disable features not implemented. Instruction fetches disabled in Debug state. For most implementations no TLB accesses take place in Debug state, and nIUL and nIUM are not required.
Unified TLB	Bits [3,1] are RAZ/WI	-
No matching control	Bits [3:2] are RAZ/WI	The TLB matching controls are not used to reduce the impact of debugging, only for advanced debugging features. If not implemented, these bits are RAZ, although the processor behaves as if they were set to 1.
TLB evictions always enabled	-	nIUL and nDUL disable TLB loading in Debug state. However TLB evictions can still take place even when these control bits are set to 0.
No loading control	Bits [1:0] are RAZ/WI	-

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

許容される実装定義の制限

DBGDSMCR は必須です。ただし、その動作には実装定義の制限を設けることができます。実装で許容される 6 つのオプションを、表 C10- 19 に示します。これらのオプションの一部は直交的です。

表 C10- 19 DBGDSCCR の動作について許容される実装定義の制限

制限	説明	注
完全な DBGDSMCR	ビット [3:0] が実装されます。	–
I-TLB 制御なし	ビット [3、1] は RAZ/WI です。	命令キャッシュのラインフィルおよび退出の非稼動化機能は実装されていません。 デバッグ状態での命令フェッチは不可能になります。ほとんどの実装では、デバック状態で TLB へのアクセスは行われず、nIUL および nIUM は必要ありません。
統一 TLB	ビット [3、1] は RAZ/WI です。	–
一致制御なし	ビット [3:2] は RAZ/WI です。	デバッグの影響を軽減するため、TLB 一致制御は使用されず、高度なデバッグ機能にのみ使用されます。実装されていない場合、これらのビットは RAZ です。ただし、プロセッサはこれらのビットが 1 にセットされているかのように動作します。
TLB の退出が常に可能	–	デバッグ状態で、nIUL および nDUL により TLB のロードが不可能になります。ただし、これらの制御ビットが 0 にセットされている場合も、TLB の退出は依然として行われる可能性があります。
ロード制御なし	ビット [1:0] は RAZ/WI です。	–

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.8 Management registers, ARMv7 only

Support for the management registers depends on the ARM architecture version:

ARMv6 These registers are not defined in ARMv6.

ARMv7 The processor identification registers are summarized in this section and are defined in one or more of:

- *CP15 registers for a VMSA implementation* on page B3-64
- *CP15 registers for a PMSA implementation* on page B4-22
- Chapter B5 *The CPUID Identification Scheme*.

Additional management registers are defined in this section.

The layout of the management registers, registers 832-1023, complies with the *CoreSight Architecture Specification*.

Processor identification registers summarizes the processor identification registers.

The following sections describe the remaining management registers:

- *Integration Mode Control Register (DBGITCTRL)* on page C10-91
- *Claim Tag Set Register (DBGCLAIMSET)* on page C10-92
- *Claim Tag Clear Register (DBGCLAIMCLR)* on page C10-93
- *Lock Access Register (DBGLAR)* on page C10-94
- *Lock Status Register (DBGLSR)* on page C10-95
- *Authentication Status Register (DBGAUTHSTATUS)* on page C10-96
- *Device Type Register (DBGDEVTYPE)* on page C10-98
- *Debug Peripheral Identification Registers (DBGPID0 to DBGPID4)* on page C10-98
- *Debug Component Identification Registers (DBGCID0 to DBGCID3)* on page C10-102

C10.8.1 Processor identification registers

The processor identification registers return the values stored in the Main ID and feature registers of the processor.

The processor identification registers are:

- debug registers 832-895, at offsets `0xD00-0xDFC`
- read-only registers.

Note

The Extended CP14 interface `MRC` and `MCR` instructions that map to these registers are UNDEFINED in User mode and UNPREDICTABLE in privileged modes. The CP15 interface must be used to access these registers.

Table C10-20 on page C10-89 lists the processor identification registers, in register number order.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C10.8 管理レジスタ（ARMv7 のみ）

管理レジスタのサポートは、ARM アーキテクチャのバージョンによって異なります。

ARMv6 ARMv6 ではこれらのレジスタは定義されていません。

ARMv7 プロセッサ識別レジスタについてはこのセクションに要約され、次のうち 1 つ以上のセクションで定義されています。

- *VMSA 実装の CP15 レジスタ* P. B3-64
- *PMSA 実装の CP15 レジスタ* P. B4-22
- 第 B5 章「*CPUID 識別方式*」

このセクションでは、追加の管理レジスタが定義されています。

管理レジスタ（レジスタ 832 ～ 1023）のレイアウトは、『*CoreSight アーキテクチャ仕様*』に準拠しています。

プロセッサ識別レジスタの要約については、*プロセッサ識別レジスタ*を参照して下さい。

他の管理レジスタについては、次のセクションで説明します。

- *統合モード制御レジスタ (DBGITCTRL)* P. C10-91
- *クレームタグ セットレジスタ (DBGCLAIMSET)* P. C10-92
- *クレームタグ クリアレジスタ (DBGCLAIMCLR)* P. C10-93
- *ロックアクセス レジスタ (DBGLAR)* P. C10-94
- *ロックステータス レジスタ (DBGLSR)* P. C10-95
- *認証ステータスレジスタ (DBGAUTHSTATUS)* P. C10-96
- *デバイスタイプ レジスタ (DBGDEVTYPE)* P. C10-98
- *デバッグペリフェラル識別レジスタ (DBGPID0 ～ DBGPID4)* P. C10-98
- *デバッグコンポーネント識別レジスタ (DBGCID0 ～ DBGCID3)* P. C10-102

C10.8.1 プロセッサ識別レジスタ

プロセッサ識別レジスタは、プロセッサのメイン ID および機能レジスタに格納されている値を返します。

プロセッサ識別レジスタには、次の特徴があります。

- デバッグレジスタ 832 ～ 895、オフセット `0xD00` ～ `0xDFC`
- 読み出し専用レジスタ

注

これらのレジスタにマッピングされる拡張 CP14 インタフェースの `MRC` および `MCR` 命令は、ユーザモードでは未定義、特権モードでは予測不能です。これらのレジスタにアクセスするには、CP15 インタフェースを使用する必要があります。

プロセッサ識別レジスタの一覧を、レジスタ番号の順に、P. C10-89 の表 C10-20 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C10-20 Processor identification registers

Register number	Access[a]	Mnemonic	Register
832	Read-only	MIDR	Main ID Register[b]
833	Read-only	CTR	Cache Type Register[b]
834	Read-only	TCMTR	TCM Type Register[b]
835	Read-only	TLBTR	TLB Type Register[b]
836	Read-only	MPUIR	MPU Type Register[b]
837	Read-only	MPIDR	Multiprocessor Affinity Register[b]
838, 839	Read-only	-	Alias of Main ID Register[b]
840	Read-only	ID_PFR0	Processor Feature Register 0
841	Read-only	ID_PFR1	Processor Feature Register 1
842	Read-only	ID_DFR0	Debug Feature Register 0
843	Read-only	ID_AFR0	Auxiliary Feature Register 0
844	Read-only	ID_MMFR0	Memory Model Feature Register 0
845	Read-only	ID_MMFR1	Memory Model Feature Register 1
846	Read-only	ID_MMFR2	Memory Model Feature Register 2
847	Read-only	ID_MMFR3	Memory Model Feature Register 3
848	Read-only	ID_ISAR0	Instruction Set Attribute Register 0
849	Read-only	ID_ISAR1	Instruction Set Attribute Register 1
850	Read-only	ID_ISAR2	Instruction Set Attribute Register 2
851	Read-only	ID_ISAR3	Instruction Set Attribute Register 3
852	Read-only	ID_ISAR4	Instruction Set Attribute Register 4
853	Read-only	ID_ISAR5	Instruction Set Attribute Register 5
854-895	-	-	Reserved, UNK/SBZP

a. For more information, see *CP14 debug registers access permissions* on page C6-36 and *Permission summaries for memory-mapped and external debug interfaces* on page C6-45.
b. Identification registers with register numbers 832-839 return the same value as an `MRC` instruction `MRC p15,0,<Rt>,c0,c0,<opc2>`, where `<opc2>` = (register number - 832).

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C10-20 プロセッサ識別レジスタ

レジスタ番号	アクセス[a]	ニーモニック	レジスタ
832	読み出し専用	MIDR	メイン ID レジスタ[b]
833	読み出し専用	CTR	キャッシュタイプ レジスタ[b]
834	読み出し専用	TCMTR	TCM タイプ レジスタ[b]
835	読み出し専用	TLBTR	TLB タイプレジスタ[b]
836	読み出し専用	MPUIR	MPU タイプレジスタ[b]
837	読み出し専用	MPIDR	マルチプロセッサ類似性レジスタ[b]
838、839	読み出し専用	–	メイン ID レジスタのエイリアス[b]
840	読み出し専用	ID_PFR0	プロセッサ機能レジスタ 0
841	読み出し専用	ID_PFR1	プロセッサ機能レジスタ 1
842	読み出し専用	ID_DFR0	デバッグ機能レジスタ 0
843	読み出し専用	ID_AFR0	補助機能レジスタ 0
844	読み出し専用	ID_MMFR0	メモリモデル機能レジスタ 0
845	読み出し専用	ID_MMFR1	メモリモデル機能レジスタ 1
846	読み出し専用	ID_MMFR2	メモリモデル機能レジスタ 2
847	読み出し専用	ID_MMFR3	メモリモデル機能レジスタ 3
848	読み出し専用	ID_ISAR0	命令セット属性レジスタ 0
849	読み出し専用	ID_ISAR1	命令セット属性レジスタ 1
850	読み出し専用	ID_ISAR2	命令セット属性レジスタ 2
851	読み出し専用	ID_ISAR3	命令セット属性レジスタ 3
852	読み出し専用	ID_ISAR4	命令セット属性レジスタ 4
853	読み出し専用	ID_ISAR5	命令セット属性レジスタ 5
854–895	–	–	予約、UNK/SBZP

a. 詳細については、*CP14 デバッグレジスタのアクセス許可* P. C6-36 と *メモリマップおよび外部デバッグインタフェースのアクセス許可の概要* P. C6-45 を参照して下さい。
b. レジスタ番号 832 ～ 839 の識別レジスタは、`MRC` 命令の `MRC p15,0,<Rt>,c0,c0,<opc2>` と同じ値を返します。ここで、`<opc2>` = (レジスタ番号 – 832) です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996–1998, 2000, 2004–2008 ARM Limited. All rights reserved.

Some of these registers form part of the CPUID scheme and are described in Chapter B5 *The CPUID Identification Scheme*. The other ARMv7 registers are described in either or both of:

- *CP15 registers for a VMSA implementation* on page B3-64
- *CP15 registers for a PMSA implementation* on page B4-22.

Table C10-21 shows where each register is described

Table C10-21 Index to descriptions of the processor Identification registers

Register	Description, VMSA	Description, PMSA
Main ID Register	*c0, Main ID Register (MIDR)* on page B3-81	*c0, Main ID Register (MIDR)* on page B4-32
Cache Type Register	*c0, Cache Type Register (CTR)* on page B3-83	*c0, Cache Type Register (CTR)* on page B4-34
TCM Type Register	*c0, TCM Type Register (TCMTR)* on page B3-85	*c0, TCM Type Register (TCMTR)* on page B4-35
TLB Type Register	*c0, TLB Type Register (TLBTR)* on page B3-86	VMSA only. Alias of Main ID Register.
MPU Type Register	PMSA only. Alias of Main ID Register.	*c0, MPU Type Register (MPUIR)* on page B4-36
Multiprocessor Affinity Register	*c0, Multiprocessor Affinity Register (MPIDR)* on page B3-87	*c0, Multiprocessor Affinity Register (MPIDR)* on page B4-37
Processor Feature Register 0 Processor Feature Register 1	*CP15 c0, Processor Feature registers* on page B5-4	
Debug Feature Register 0	*c0, Debug Feature Register 0 (ID_DFR0)* on page B5-6	
Auxiliary Feature Register 0	*c0, Auxiliary Feature Register 0 (ID_AFR0)* on page B5-8	
Memory Model Feature Register 0 to Memory Model Feature Register 3	*CP15 c0, Memory Model Feature registers* on page B5-9	
Instruction Set Attribute Register 0 to Instruction Set Attribute Register 5	*CP15 c0, Instruction Set Attribute registers* on page B5-19	

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

これらのレジスタの一部は、CPUID 方式の一部を構成し、第 B5 章*「CPUID 識別方式」*に説明されています。他の ARMv7 レジスタについては、次のいずれかまたは両方に説明されています。

- *VMSA 実装の CP15 レジスタ* P. B3-64
- *PMSA 実装の CP15 レジスタ* P. B4-22

各レジスタについて説明されている場所を、表 C10-21 に示します。

表 C10-21 プロセッサ識別レジスタの説明の索引

レジスタ	説明 (VMSA)	説明 (PMSA)
メイン ID レジスタ	*c0、メイン ID レジスタ (MIDR)* P. B3-81	*c0、メイン ID レジスタ (MIDR)* P. B4-32
キャッシュタイプ レジスタ	*c0、キャッシュタイプ レジスタ (CTR)* P. B3-83	*c0、キャッシュタイプ レジスタ (CTR)* P. B4-34
TCM タイプレジスタ	*c0、TCM タイプレジスタ (TCMTR)* P. B3-85	*c0、TCM タイプレジスタ (TCMTR)* P. B4-35
TLB タイプレジスタ	*c0、TLB タイプレジスタ (TLBTR)* P. B3-86	VMSA のみ。メイン ID レジスタのエイリアス
MPU タイプレジスタ	PMSA のみ。メイン ID レジスタのエイリアス	*c0、MPU タイプレジスタ (MPUIR)* P. B4-36
マルチプロセッサ類似性レジスタ	*c0、マルチプロセッサ類似性レジスタ (MPIDR)* P. B3-87	*c0、マルチプロセッサ類似性レジスタ (MPIDR)* P. B4-37
プロセッサ機能レジスタ 0 プロセッサ機能レジスタ 1	*CP15 c0、プロセッサ機能レジスタ* P. B5-4	
デバッグ機能レジスタ 0	*c0、デバッグ機能レジスタ 0 (ID_DFR0)* P. B5-6	
補助機能レジスタ 0	*c0、補助機能レジスタ 0 (ID_AFR0)* P. B5-8	
メモリモデル機能レジスタ 0 ～ 3	*CP15 c0、メモリモデル機能レジスタ* P. B5-9	
命令セット属性レジスタ 0 ～ 5	*CP15 c0、命令セット属性レジスタ* P. B5-19	

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.8.2 Integration Mode Control Register (DBGITCTRL)

The Integration Mode Control Register, DBGITCTRL, enables the device to switch from its default functional mode into *integration mode*, where the inputs and outputs of the device can be directly controlled for integration testing or topology detection. When the processor is in integration mode, the IMPLEMENTATION DEFINED integration registers can be used to drive output values and to read inputs.

The DBGITCTRL Register is:

- debug register 960, at offset `0xF00`
- a read/write register
- when the Security Extensions are implemented, a Common register.

The format of the DBGITCTRL Register is:

Bits [31:1] Reserved, UNK/SBZP.

Integration mode enable, bit [0]

The possible values of this bit are:

0 Normal operation

1 Integration mode enabled.

When this bit is set to 1, the device reverts to an integration mode to enable integration testing or topology detection. The integration mode behavior is IMPLEMENTATION DEFINED.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.8.2 統合モード制御レジスタ (DBGITCTRL)

統合モード制御レジスタ (DBGITCTRL) は、デバイスをデフォルトの機能モードから*統合モード*に切り替えるために使用されます。統合モードでは、デバイスの入力と出力を直接制御し、統合テストやトポロジ検出を行うことができます。プロセッサが統合モードのとき、実装定義の統合レジスタを使用して、出力値の駆動と入力の読み出しを実行できます。

DBGITCTRL レジスタには、次の特徴があります。

- デバッグレジスタ 960、オフセット `0xF00`
- 読み出し / 書き込みレジスタ
- セキュリティ拡張機能が実装されている場合、共通レジスタである。

DBGITCTRL レジスタのフォーマットは次のとおりです。

ビット [31:1] 予約、UNK/SBZP

統合モードイネーブル、ビット [0]

このビットの可能な値は次のとおりです。

0 通常動作

1 統合モードが稼動しています。

このビットが 1 にセットされているとき、デバイスは統合モードに切り替わり、統合テストやトポロジ検出が可能になります。統合モードの動作は実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.8.3 Claim Tag Set Register (DBGCLAIMSET)

The Claim Tag Set Register, DBGCLAIMSET, enables the CLAIM bits, bits [7:0] of the register, to be set to 1. CLAIM bits do not have any specific functionality. ARM expects the usage model to be that an external debugger and a debug monitor can set specific bits to 1 to claim the corresponding debug resources.

The CLAIM bits are always RAO in the DBGCLAIMSET Register. This enables a debugger to identify the number of CLAIM bits that are implemented. See *Claim Tag Clear Register (DBGCLAIMCLR)* on page C10-93 for details of how to:

- clear CLAIM bits to 0
- read the current values of the CLAIM bits.

The DBGCLAIMSET Register is:

- debug register 1000, at offset `0xFA0`
- a read/write register, in which:
  - — the CLAIM bits are always RAO
  - — writing 0 to a CLAIM bit has no effect
- when the Security Extensions are implemented, a Common register.

The format of the DBGCLAIMSET Register is:

31 … 8	7 … 0
Reserved, RAZ/SBZP	CLAIM

Bits [31:8] Reserved, RAZ/SBZP.

CLAIM bits, bits [7:0]

Writing a 1 to one of these bits sets the corresponding CLAIM bit to 1. Multiple bits can be set to 1 in a single write operation.

Writing 0 to one of these bits has no effect.

You must use the DBGCLAIMCLR Register to:

- read the values of the CLAIM bits
- clear a CLAIM bit to 0.

These bits are always RAO.

If a debugger reads this register, the bits that are set to 1 correspond to the implemented CLAIM bits.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C10.8.3 クレームタグ セットレジスタ (DBGCLAIMSET)

クレームタグ セットレジスタ (DBGCLAIMSET) は、レジスタのビット [7:0] である CLAIM ビットを 1 にセットできるようにします。CLAIM ビットには、特別な機能はありません。外部デバッガとデバッグモニタが、特定のビットを 1 にセットすることで、対応するデバッグリソースを要求できるような使用モデルを想定しています。

CLAIM ビットは、DBGCLAIMSET レジスタでは常に RAO です。これによって、デバッガは実装されている CLAIM ビットの数を識別できます。次の方法の詳細については、*クレームタグ クリアレジスタ (DBGCLAIMCLR)* P. C10-93 を参照して下さい。

- CLAIM ビットを 0 にクリアする。
- CLAIM ビットの現在の値を読み出す。

DBGCLAIMSET レジスタには、次の特徴があります。

- デバッグレジスタ 1000、オフセット `0xFA0`
- 読み出し / 書き込みレジスタで、次の特徴があります。
  - ー CLAIM ビットは常に RAO である。
  - ー CLAIM ビットに対する 0 の書き込みは無効である。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

DBGCLAIMSET レジスタのフォーマットは次のとおりです。

31 … 8	7 … 0
予約、RAZ/SBZP	CLAIM

ビット [31:8] 予約、RAZ/SBZP

CLAIM ビット、ビット [7:0]

これらのビットの 1 つに 1 を書き込むと、対応する CLAIM ビットが 1 にセットされます。単一の書き込み操作で、複数のビットを 1 にセットすることが可能です。

これらのビットのいずれかに対する 0 の書き込みは無効です。

次の目的には、DBGCLAIMCLR レジスタを使用する必要があります。

- CLAIM ビットの値を読み出す。
- CLAIM ビットを 0 にクリアする。

これらのビットは常に RAO です。

デバッガがこのレジスタを読み出すと、1 にセットされているビットは、実装されている CLAIM ビットに対応しています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.8.4 Claim Tag Clear Register (DBGCLAIMCLR)

The Claim Tag Clear Register, DBGCLAIMCLR, enables the values of the CLAIM bits, bits [7:0] of the register, to be:

- read
- cleared to 0.

For more information about the CLAIM bits and how they might be used, see *Claim Tag Set Register (DBGCLAIMSET)* on page C10-92.

The DBGCLAIMCLR Register is:

- debug register 1001, at offset `0xFA4`
- a read/write register, in which:
  - — writing 0 to a CLAIM bit has no effect
  - — writing 1 to a CLAIM bit clears that bit to 0
  - — a read of the register returns the current values of the CLAIM bits
- when the Security Extensions are implemented, a Common register.

The format of the DBGCLAIMCLR Register is:

31 … 8	7 … 0
Reserved, RAZ/SBZP	CLAIM

Bits [31:8] Reserved, RAZ/SBZP.

CLAIM bits, bits [7:0]

Writing a 1 to one of these bits clears the corresponding CLAIM bit to 0. Multiple bits can be cleared to 0 in a single write operation.

Writing 0 to one of these bits has no effect.

Reading the register returns the current values of these bits.

The debug logic reset value of each of these bits is 0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.8.4 クレームタグ クリアレジスタ (DBGCLAIMCLR)

クレームタグ クリアレジスタ (DBGCLAIMCLR) は、レジスタの CLAIM ビット（ビット [7:0]）の値に対して次の操作を行うために使用します。

- 読み出し
- 0 にクリア

CLAIM ビットの詳細と使用法については、*クレームタグ セットレジスタ (DBGCLAIMSET)* P. C10- 92 を参照して下さい。

DBGCLAIMCLR レジスタには、次の特徴があります。

- デバッグレジスタ 1001、オフセット `0xFA4`
- 読み出し / 書き込みレジスタで、次の特徴があります。
  - – CLAIM ビットに対する 0 の書き込みは無効です。
  - – CLAIM ビットに 1 を書き込むと、そのビットが 0 にクリアされる。
  - – レジスタを読み出すと、CLAIM ビットの現在の値が返される。
- セキュリティ拡張機能が実装されている場合、共通レジスタである。

DBGCLAIMCLR レジスタのフォーマットは次のとおりです。

31 … 8	7 … 0
予約、RAZ/SBZP	CLAIM

ビット [31:8] 予約、RAZ/SBZP

CLAIM ビット、ビット [7:0]

これらのビットの 1 つに 0 を書き込むと、対応する CLAIM ビットが 0 にクリアされます。単一の書き込み操作で、複数のビットを 0 にクリアすることが可能です。

これらのビットのいずれかに対する 0 の書き込みは無効です。

レジスタを読み出すと、これらのビットについて現在の値が返されます。

デバッグロジック リセット時の各ビットの値は 0 です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.8.5 Lock Access Register (DBGLAR)

The Lock Access Register, DBGLAR, provides a lock on writes to the debug registers through the memory-mapped interface. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

You must use the DBGLSR to check the current status of the lock, see *Lock Status Register (DBGLSR)* on page C10-95.

The DBGLAR is:

- debug register 1004, at offset `0xFB0`
- a write-only register
- only defined in the memory-mapped interface
- when the Security Extensions are implemented, a Common register.

Note

- Debug register 1004, at offset `0xFB0`, is reserved in both the Extended CP14 interface and the external debug interface.
- Do not confuse the Software Lock mechanism with the OS Lock described in *The OS Save and Restore mechanism* on page C6-8.

The format of the DBGLAR is:

31 … 0
Lock Access control

Lock Access control, bits [31:0]

Writing the key value `0xC5ACCE55` to this field clears the lock, enabling write accesses to the debug registers through the memory-mapped interface.

Writing any other value to this register sets the lock, disabling write accesses to the debug registers through the memory-mapped interface.

Note

- In implementations with separate core and debug power-domains, this lock is maintained in the debug power domain. Its state is unaffected by the core power domain powering down.
- This lock is set on debug logic reset, that is, on a **PRESETDBGn** or **nSYSPORESET** reset.

Accesses through the memory-mapped interface to locked debug registers are ignored. For more information, see *Permissions in relation to locks* on page C6-27.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C10.8.5 ロックアクセス レジスタ (DBGLAR)

ロックアクセス レジスタ (DBGLAR) は、メモリマップされたインタフェース経由でのデバッグレジスタへの書き込みに対して、ロックを提供します。このロック機構を使用すると、デバッグレジスタの内容が誤って破損する危険を減らすことができます。ただし、過失または悪意による破損を完全に防止することはできません。

現在のロック状態をチェックするには、DBGLSR を使用する必要があります。*ロックステータス レジスタ (DBGLSR)* P. C10-95 を参照して下さい。

DBGLAR には、次の特徴があります。

- デバッグレジスタ 1004、オフセット `0xFB0`
- 書き込み専用レジスタ
- メモリマップされたインタフェースでのみ定義される。
- セキュリティ拡張機能が実装されている場合、共通レジスタである。

注

- デバッグレジスタ 1004、オフセット `0xFB0` は、拡張 CP14 インタフェースおよび外部デバッグインタフェースでは予約されています。
- ソフトウェアロック機構を、*OS 保存および復元機構* P. C6-8 で説明されている OS ロックと混同しないで下さい。

DBGLAR のフォーマットは次のとおりです。

31 … 0
ロックアクセス制御

ロックアクセス制御、ビット [31:0]

このフィールドにキー値 `0xC5ACCE55` を書き込むとロックがクリアされ、メモリマップされたインタフェース経由でデバッグレジスタに書き込みアクセスが可能になります。

このレジスタに他の値を書き込むとロックがセットされ、メモリマップされたインタフェース経由でデバッグレジスタに書き込みアクセスが不可能になります。

注

- コアとデバッグの電力ドメインが独立している実装では、このロックはデバッグ電力ドメインで保守されます。この状態は、コア電力ドメインの電力がオフになっても影響を受けません。
- このロックは、デバッグロジック リセット時、つまり PRESETDBGn または nSYSPORESET リセット時にセットされます。

メモリマップされたインタフェース経由で、ロックされたデバッグレジスタにアクセスした場合、無視されます。詳細については、*ロックとアクセス許可との関係* P. C6-26 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.8.6 Lock Status Register (DBGLSR)

The Lock Status Register, DBGLSR, provides status information for the debug registers lock. For more information about this lock see *Lock Access Register (DBGLAR)* on page C10-94.

The DBGLSR is:

- debug register 1005, at offset `0xFB4`
- a read-only register
- only defined in the memory-mapped interface
- when the Security Extensions are implemented, a Common register.

Note

Debug register 1005, at offset `0xFB4`, is reserved in both the Extended CP14 interface and the external debug interface.

The format of the DBGLSR is:

Bits [31:3] Reserved, UNK.

32-bit access, bit [2]

This bit is always RAZ. It indicates that a 32-bit access is needed to write the key to the Lock Access Register.

Locked, bit [1]

This bit indicates the status of the debug registers lock. The possible values are:

0 Lock clear. Debug register writes are permitted.

1 Lock set. Debug register writes are ignored.

The debug registers lock is set or cleared by writing to the DBGLAR, see *Lock Access Register (DBGLAR)* on page C10-94.

The debug logic reset value of this bit is 1.

Lock implemented, bit [0]

This bit is RAO.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.8.6 ロックステータス レジスタ (DBGLSR)

ロックステータス レジスタ (DBGLSR) は、デバッグレジスタ ロックのステータス情報を提供します。このロックの詳細については、*ロックアクセス レジスタ (DBGLAR)* P. C10-94 を参照して下さい。

DBGLSR には、次の特徴があります。

- デバッグレジスタ 1005、オフセット `0xFB4`
- 読み出し専用レジスタ
- メモリマップされたインタフェースでのみ定義される。
- セキュリティ拡張機能が実装されている場合、共通レジスタである。

注

デバッグレジスタ 1005、オフセット `0xFB4` は、拡張 CP14 インタフェースおよび外部デバッグインタフェースでは予約されています。

DBGLSR のフォーマットは次のとおりです。

ビット [31:3] 予約、UNK

32 ビットアクセス、ビット [2]

このビットは常に RAZ です。ロックアクセス レジスタへキーを書き込むため、32 ビットアクセスが必要なことを示しています。

ロック、ビット [1]

このビットは、デバッグレジスタ ロックのステータスを示します。可能な値は次のとおりです。

0 ロックがクリアされています。デバッグレジスタへの書き込みが許可されます。

1 ロックがセットされています。デバッグレジスタへの書き込みは無視されます。

デバッグレジスタ ロックは、DBGLAR への書き込みによりセットまたはクリアされます。*ロックアクセス レジスタ (DBGLAR)* P. C10-94 を参照して下さい。

デバッグロジック リセット時のこのビットの値は 1 です。

ロック実装、ビット [0]

このビットは RAO です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.8.7 Authentication Status Register (DBGAUTHSTATUS)

The Authentication Status Register, DBGAUTHSTATUS, indicates the implemented debug features and provides the current values of the configuration inputs that determine the debug permissions. The value returned depends on whether the processor implements the Security Extensions.

The DBGAUTHSTATUS Register is:

- debug register 1006, at offset `0xFB8`
- a read-only register
- when the Security Extensions are implemented, a Common register.

When the Security Extensions are implemented, the format of the DBGAUTHSTATUS Register is:

Bits [31:8] Reserved, UNK.

Secure non-invasive debug features implemented, bit [7]

This bit is RAO, Secure non-invasive debug features are implemented.

Secure non-invasive debug enabled, bit [6]

This bit indicates the logical result of:

(**DBGEN** OR **NIDEN**) AND (**SPIDEN** OR **SPNIDEN**).

Secure invasive debug features implemented, bit [5]

This bit is RAO, Secure invasive debug features are implemented.

Secure invasive debug enabled, bit [4]

This bit indicates the logical result of (**DBGEN** AND **SPIDEN**).

Non-secure non-invasive debug features implemented, bit [3]

This bit is RAO, Non-secure non-invasive debug features are implemented.

Non-secure non-invasive debug enabled, bit [2]

This bit indicates the logical result of (**DBGEN** OR **NIDEN**)

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.8.7 認証ステータスレジスタ (DBGAUTHSTATUS)

認証ステータスレジスタ (DBGAUTHSTATUS) は、デバッグ機能が実装されていることを示し、デバッグ許可を決定する構成入力の現在の値を提供します。返される値は、プロセッサにセキュリティ拡張機能が実装されているかどうかによって異なります。

DBGAUTHSTATUS レジスタには、次の特徴があります。

- デバッグレジスタ 1006、オフセット `0xFB8`
- 読み出し専用レジスタ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

セキュリティ拡張機能が実装されている場合の DBGAUTHSTATUS レジスタのフォーマットは次のとおりです。

ビット [31:8] 予約、UNK

セキュア非侵襲性デバッグ機能の実装、ビット [7]

このビットは RAO で、セキュア非侵襲性デバッグ機能が実装されていることを示します。

セキュア非侵襲性デバッグイネーブル、ビット [6]

このビットは、次の論理式の結果を示しています。

(DBGEN OR NIDEN) AND (SPIDEN OR SPNIDEN)

セキュア侵襲性デバッグ機能の実装、ビット [5]

このビットは RAO で、セキュア侵襲性デバッグ機能が実装されていることを示します。

セキュア侵襲性デバッグイネーブル、ビット [4]

このビットは、論理式 (DBGEN AND SPIDEN) の結果を示しています。

非セキュア非侵襲性デバッグ機能の実装、ビット [3]

このビットは RAO で、非セキュア非侵襲性デバッグ機能が実装されていることを示します。

非セキュア非侵襲性デバッグイネーブル、ビット [2]

このビットは、論理式 (DBGEN OR NIDEN) の結果を示しています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Non-secure invasive debug features implemented, bit [1]

This bit is RAO, Non-secure invasive debug features are implemented.

Non-secure invasive debug enabled, bit [0]

This bit indicates the logical state of the **DBGEN** signal.

When the Security Extensions are not implemented, the format of the DBGAUTHSTATUS Register is:

Bits [31:8] Reserved, UNK.

Secure non-invasive debug features implemented, bit [7]

This bit is RAO, Secure non-invasive debug features are implemented.

Secure non-invasive debug enabled, bit [6]

This bit indicates the logical result of (**DBGEN** OR **NIDEN**).

Secure invasive debug features implemented, bit [5]

This bit reads is RAO, Secure invasive debug features are implemented.

Secure invasive debug enabled, bit [4]

This bit indicates the logical state of the **DBGEN** signal.

Non-secure non-invasive debug features implemented, bit [3]

This bit is RAZ, Non-secure non-invasive debug features are not implemented.

Non-secure non-invasive debug enabled, bit [2]

This bit is RAZ.

Non-secure invasive debug features implemented, bit [1]

This bit is RAZ, Non-secure invasive debug features are not implemented.

Non-secure invasive debug enabled, bit [0]

This bit is RAZ.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

非セキュア侵襲性デバッグ機能の実装、ビット [1]

このビットは RAO で、非セキュア侵襲性デバッグ機能が実装されていることを示します。

非セキュア侵襲性デバッグイネーブル、ビット [0]

このビットは、DBGEN 信号の論理的な状態を示しています。

セキュリティ拡張機能が実装されていない場合の DBGAUTHSTATUS レジスタのフォーマットは次のとおりです。

ビット [31:8] 予約、UNK

セキュア非侵襲性デバッグ機能の実装、ビット [7]

このビットは RAO で、セキュア非侵襲性デバッグ機能が実装されていることを示します。

セキュア非侵襲性デバッグイネーブル、ビット [6]

このビットは、論理式 (DBGEN OR NIDEN) の結果を示しています。

セキュア侵襲性デバッグ機能の実装、ビット [5]

このビットは RAO で、セキュア侵襲性デバッグ機能が実装されていることを示します。

セキュア侵襲性デバッグイネーブル、ビット [4]

このビットは、DBGEN 信号の論理的な状態を示しています。

非セキュア非侵襲性デバッグ機能の実装、ビット [3]

このビットは RAZ で、非セキュア非侵襲性デバッグ機能が実装されていないことを示します。

非セキュア非侵襲性デバッグイネーブル、ビット [2]

このビットは RAZ です。

非セキュア侵襲性デバッグ機能の実装、ビット [1]

このビットは RAZ で、非セキュア侵襲性デバッグ機能が実装されていないことを示します。

非セキュア侵襲性デバッグイネーブル、ビット [0]

このビットは RAZ です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If a processor does not implement the Security Extensions, it does not implement any Non-secure debug features.

C10.8.8 Device Type Register (DBGDEVTYPE)

The Device Type Register, DBGDEVTYPE, provides the CoreSight device type information for the Debug architecture. The DBGDEVTYPE register must be implemented in all CoreSight components, and indicates the type of debug component.

The DBGDEVTYPE Register is:

- debug register 1011, at offset `0xFCC`
- a read-only register
- when the Security Extensions are implemented, a Common register.

The format of the DBGDEVTYPE Register is:

Bits [31:8] Reserved, RAZ.

Sub type, bits [7:4]

This field reads as `0x1`, indicating a processor.

Main class, bits [3:0]

This field reads as `0x5`, indicating Debug logic.

For more information about the CoreSight registers see the *CoreSight Architecture Specification*.

C10.8.9 Debug Peripheral Identification Registers (DBGPID0 to DBGPID4)

The Debug Peripheral Identification Registers provide standard information required by all components that conform to the ARM Debug Interface v5 specification. They identify a peripheral in a particular namespace. For more information, see the *ARM Debug Interface v5 Architecture Specification*.

The Debug Peripheral Identification Registers are:

- debug registers 1012-1019, at offsets `0xFD0-0xFEC`
- read-only registers.
- when the Security Extensions are implemented, Common registers.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

プロセッサにセキュリティ拡張機能が実装されていない場合、非セキュアデバッグ機能は一切実装されていません。

C10.8.8 デバイスタイプ レジスタ (DBGDEVTYPE)

デバイスタイプ レジスタ (DBGDEVTYPE) は、デバッグアーキテクチャの CoreSight デバイスタイプ情報を提供します。DBGDEVTYPE レジスタは、すべての CoreSight コンポーネントに実装され、デバッグコンポーネントのタイプを示す必要があります。

DBGDEVTYPE レジスタには、次の特徴があります。

- デバッグレジスタ 1011、オフセット `0xFCC`
- 読み出し専用レジスタ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

DBGDEVTYPE レジスタのフォーマットは次のとおりです。

ビット [31:8] 予約、RAZ

サブタイプ、ビット [7:4]

このフィールドは `0x1` として読み出され、プロセッサを示します。

メインクラス、ビット [3:0]

このフィールドは `0x5` として読み出され、デバッグロジックを示します。

CoreSight レジスタの詳細については、『*CoreSight アーキテクチャ仕様*』を参照して下さい。

C10.8.9 デバッグペリフェラル識別レジスタ (DBGPID0 ～ DBGPID4)

デバッグペリフェラル識別レジスタは、ARM デバッグインタフェース v5 仕様に準拠する、すべてのコンポーネントに必須である標準の情報を提供します。これらの情報は、特定の名前空間でペリフェラルを識別します。詳細については、『*ARM デバッグインタフェース v5、アーキテクチャ仕様書*』を参照して下さい。

デバッグペリフェラル識別レジスタには、次の特徴があります。

- デバッグレジスタ 1012 ～ 1019、オフセット `0xFD0` ～ `0xFEC`
- 読み出し専用レジスタ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

- ARMv7 only defines Debug Peripheral ID Registers 0 to 4, and reserves space for Debug Peripheral ID Registers 5 to 7.
- The register number order of the Debug Peripheral ID Registers does not match the numerical order ID0 to ID7, see Table C10-22.

Table C10-22 lists the Debug Peripheral Identification Registers in register number order.

Table C10-22 Debug Peripheral Identification Registers

Register: Number	Offset	Description	Reference
1012	0xFD0	Debug Peripheral ID4	*DBGPID4* on page C10-102
1013	0xFD4	Reserved for Debug Peripheral ID5, DBGPID5	-
1014	0xFD8	Reserved for Debug Peripheral ID6, DBGPID6	-
1015	0xFDC	Reserved for Debug Peripheral ID7, DBGPID7	-
1016	0xFE0	Debug Peripheral ID0	*DBGPID0* on page C10-101
1017	0xFE4	Debug Peripheral ID1	*DBGPID1* on page C10-101
1018	0xFE8	Debug Peripheral ID2	*DBGPID2* on page C10-101
1019	0xFEC	Debug Peripheral ID3	*DBGPID0* on page C10-101

Only bits [7:0] of each Debug Peripheral ID Register are used. This means that the format of each register is:

31 … 8	7 … 0
Reserved, RAZ	Peripheral ID data

The eight Debug Peripheral ID Registers can be considered as defining a single 64-bit Peripheral ID, as shown in Figure C10-1.

Figure C10-1 Mapping between Debug Peripheral ID Registers and a 64-bit Peripheral ID value

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

- ARMv7 では、デバッグペリフェラル識別レジスタ 0 ～ 4 のみが定義され、5 ～ 7 の空間は予約されています。
- デバッグペリフェラル識別レジスタのレジスタ番号の順序は、ID0 ～ ID7 の数値的な順序とは一致しません。表 C10-22 を参照して下さい。

デバッグペリフェラル識別レジスタの一覧を、レジスタ番号の順に、表 C10-22 に示します。

表 C10-22 デバッグペリフェラル識別レジスタ

レジスタ 番号	オフセット	説明	参照先
1012	0xFD0	デバッグペリフェラル識別レジスタ 4	*DBGPID4* P. C10-102
1013	0xFD4	デバッグペリフェラル識別レジスタ 5 (DBGPID5) 用に予約	–
1014	0xFD8	デバッグペリフェラル識別レジスタ 6 (DBGPID6) 用に予約	–
1015	0xFDC	デバッグペリフェラル識別レジスタ 7 (DBGPID7) 用に予約	–
1016	0xFE0	デバッグペリフェラル識別レジスタ 0	*DBGPID0* P. C10-101
1017	0xFE4	デバッグペリフェラル識別レジスタ 1	*DBGPID1* P. C10-101
1018	0xFE8	デバッグペリフェラル識別レジスタ 2	*DBGPID2* P. C10-101
1019	0xFEC	デバッグペリフェラル識別レジスタ 3	*DBGPID0* P. C10-101

各デバッグペリフェラル識別レジスタについて、ビット [7:0] のみが使用されます。各レジスタのフォーマットは次のとおりです。

31 … 8	7 … 0
予約、RAZ	ペリフェラル ID データ

8 つのデバッグペリフェラル識別レジスタは、図 C10-1 に示すように、単一の 64 ビットペリフェラル ID を定義するものと考えることができます。

図 C10-1 デバッグペリフェラル識別レジスタと 64 ビットのペリフェラル ID 値とのマッピング

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure C10-2 shows the fields in the 64-bit Peripheral ID value, and includes the field values for fields that:

- have fixed values, including the bits that are reserved, RAZ
- have fixed values in a device that is designed by ARM.

For more information about the fields and their values see Table C10-23.

Bits with no value shown are IMPLEMENTATION DEFINED
Some bit values shown are for a device designed by ARM Limited. See text for details.

Figure C10-2 Peripheral ID fields, with values for a design by ARM

Table C10-23 shows the fields in the Peripheral ID.

Table C10-23 Fields in the Debug Peripheral Identification Registers

Name	Size	Description	Register
4KB count	4 bits	Log_2 of the number of 4KB blocks occupied by the device. In v7 Debug, the debug registers occupy a single 4KB block, so this field is always `0x0`.	DBGPID4
JEP 106 code	4+7 bits	Identifies the designer of the device. This value consists of: a 4-bit continuation code, also described as the bank number a 7-bit identity code. For implementations designed by ARM, the continuation code is `0x4` (bank 5), and the identity code is `0x3B`. For more information, see *JEP106, Standard Manufacturers Identification Code*.	DBGPID1, DBGPID2, DBGPID4
RevAnd	4 bits	Manufacturing Revision Number. Indicates a late modification to the device, usually as a result of an Engineering Change Order. This field starts at `0x0` and is incremented by the integrated circuit manufacturer on metal fixes.	DBGPID3
Customer modified	4 bits	Indicates an endorsed modification to the device. If the system designer cannot modify the RTL supplied by the processor designer then this field is RAZ.	DBGPID3

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

図 C10-2 に示されているのは、64 ビットのペリフェラル ID 値のフィールドで、次のフィールドの値が含まれています。

- 固定の値を持つフィールド。これには、予約されていて RAZ のビットも含まれています。
- ARM によって設計されたデバイスで、固定の値を持つフィールド

これらのフィールドとその値の詳細については、表 C10-23 を参照して下さい。

ここに値が示されていないビットは実装定義です。
ここに示されているビット値の一部は、ARM Limitedで設計されたデバイス用です。詳細については、本文を参照して下さい。

図 C10-2 ARM による設計に対応する値を持つペリフェラル ID フィールド

ペリフェラル ID のフィールドを、表 C10-23 に示します。

表 C10-23 デバッグペリフェラル識別レジスタのフィールド

名前	サイズ	説明	レジスタ
4KB カウント	4 ビット	デバイスにより占有される 4KB ブロック数の Log_2。v7 デバッグでは、デバッグレジスタは単一の 4KB ブロックを占有するため、このフィールドは常に `0x0` です。	DBGPID4
JEP 106 コード	4 + 7 ビット	デバイスの設計者を示します。この値は、次のもので構成されます。 4 ビットの継続コード。バンク番号とも呼ばれます。 7 ビットの識別コード。 ARM により設計された実装では、継続コードは `0x4`（バンク 5）、識別コードは `0x3B` です。 詳細については、『*JEP106、標準製造業者識別コード*』を参照して下さい。	DBGPID1、DBGPID2、DBGPID4
RevAnd	4 ビット	製造リビジョン番号。デバイスに、通常はエンジニアリング変更指示の結果として、後で加えられた変更を示します。 このフィールドは最初は `0x0` で、集積回路が修正されるごとに、製造業者によりインクリメントされます。	DBGPID3
顧客による変更	4 ビット	デバイスに対して承認された変更を示します。 システム設計者が、プロセッサ設計者により供給された RTL を変更できない場合、このフィールドは RAZ です。	DBGPID3

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C10-23 Fields in the Debug Peripheral Identification Registers (continued)

Name	Size	Description	Register
Revision	4 bits	Revision number for the device. Starts at `0x0` and increments by 1 at both major and minor revisions.	DBGPID2
Uses JEP 106 ID code	1 bit	This bit is set to 1 when a JEP 106 Identity Code is used. This bit must be 1 on all ARMv7 implementations.	DBGPID2
Part Number	12 bits	Part number for the device. Each organization designing devices to the ARM Debug architecture specification keeps its own part number list.	DBGPID0, DBGPID1

For more information about these fields, see the *ARM Debug Interface v5 Architecture Specification*.

The following subsections describe the formats of each of the implemented Debug Peripheral ID Registers.

DBGPID0

DBGPID0 is debug register 1016 at offset `0xFE0`. Its format is:

Bits [31:8] Reserved, RAZ.

Part number[7:0], bits [7:0]

Bits [7:0] of the IMPLEMENTATION DEFINED Part number.

DBGPID1

DBGPID1 is debug register 1017 at offset `0xFE4`. Its format is:

Bits [31:8] Reserved, RAZ.

JEP Identity Code[3:0], bits [7:4]

Bits [3:0] of the IMPLEMENTATION DEFINED JEP Identity Code.

For a device designed by ARM the JEP Identity Code is `0x3B` and therefore this field is `0xB`.

Part number[11:8], bits [3:0]

Bits [11:8] of the IMPLEMENTATION DEFINED Part number.

DBGPID2

DBGPID2 is debug register 1018 at offset `0xFE8`. Its format is:

Bits [31:8] Reserved, RAZ.

Revision, bits [7:4]

The IMPLEMENTATION DEFINED revision number for the device.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 C10-23 デバッグペリフェラル識別レジスタのフィールド（続き）

名前	サイズ	説明	レジスタ
リビジョン	4ビット	デバイスのリビジョン番号。 最初は `0x0` で、メジャーおよびマイナーリビジョンの両方について 1 ずつインクリメントされます。	DBGPID2
JEP 106 ID コードの使用	1 ビット	このビットは、JEP 106 ID コードが使用されているときに 1 にセットされます。 すべての ARMv7 実装で 1 の必要があります。	DBGPID2
部品番号	12ビット	デバイスの部品番号。ARM デバッグアーキテクチャ仕様に従ってデバイスを設計する各組織が、独自の部品番号リストを保持しています。	DBGPID0、DBGPID1

これらのフィールドの詳細については、『*ARM デバッグインタフェース v5、アーキテクチャ仕様書*』を参照して下さい。

以下のサブセクションでは、実装されているデバッグペリフェラル識別レジスタそれぞれのフォーマットについて説明します。

DBGPID0

DBGPID0 はデバッグレジスタ 1016、オフセット `0xFE0` です。フォーマットは次のとおりです。

ビット [31:8] 予約、RAZ

部品番号 [7:0]、ビット [7:0]

実装定義の部品番号のビット [7:0]。

DBGPID1

DBGPID1 はデバッグレジスタ 1017、オフセット `0xFE4` です。フォーマットは次のとおりです。

ビット [31:8] 予約、RAZ

JEP ID コード [3:0]、ビット [7:4]

実装定義の JEP ID コードのビット [3:0]。

ARM で設計されたデバイスの場合、JEP ID コードは `0x3B` なので、このフィールドは `0xB` です。

部品番号 [11:8]、ビット [3:0]

実装定義の部品番号のビット [11:8]。

DBGPID2

DBGPID2 はデバッグレジスタ 1018、オフセット `0xFE8` です。フォーマットは次のとおりです。

ビット [31:8] 予約、RAZ

リビジョン、ビット [7:4]

実装定義のデバイスのリビジョン番号。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Uses JEP Code, bit [3]

For an ARMv7 implementation this bit must be one, indicating that the Peripheral ID uses a JEP 106 Identity Code.

JEP Identity Code[6:4], bits [2:0]

Bits [6:4] of the IMPLEMENTATION DEFINED JEP Identity Code.

For a device designed by ARM the JEP Identity Code is `0x3B` and therefore this field is 0b011.

DBGPID3

DBGPID3 is debug register 1019 at offset `0xFEC`. Its format is:

Bits [31:8] Reserved, RAZ.

RevAnd, bits [7:4]

The IMPLEMENTATION DEFINED manufacturing revision number for the device.

Customer modified, bits [3:0]

An IMPLEMENTATION DEFINED value that indicates an endorsed modification to the device.

If the system designer cannot modify the RTL supplied by the processor designer then this field is RAZ.

DBGPID4

DBGPID4 is debug register 1012 at offset `0xFD0`. Its format is:

Bits [31:8] Reserved, RAZ.

4KB count, bits [7:4]

This field is RAZ for all ARMv7 implementations.

JEP 106 Continuation code, bits [3:0]

The IMPLEMENTATION DEFINED JEP 106 Continuation code.

For a device designed by ARM this field is `0x4`.

C10.8.10 Debug Component Identification Registers (DBGCID0 to DBGCID3)

The Debug Component Identification Registers identify the processor as an ARM Debug Interface v5 Component. For more information, see the *ARM Debug Interface v5 Architecture Specification*.

The Debug Component Identification Registers:

- are debug registers 1020-1023, at offsets `0xFF0-0xFFC`
- are read-only registers

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

JEP コードの使用、ビット [3]

ARMv7 実装では、このビットは必ず 1 で、ペリフェラル ID に JEP 106 ID コードが使用されていることを示します。

JEP ID コード [6:4]、ビット [2:0]

実装定義の JEP ID コードのビット [6:4]。

ARM で設計されたデバイスの場合、JEP ID コードは `0x3B` なので、このフィールドは 0b011 です。

DBGPID3

DBGPID3 はデバッグレジスタ 1019、オフセット `0xFEC` です。フォーマットは次のとおりです。

ビット [31:8] 予約、RAZ

RevAnd、ビット [7:4]

デバイスの、実装定義の製造リビジョン番号。

顧客による変更、ビット [3:0]

デバイスに対して承認された変更を示す、実装定義の値。

システム設計者が、プロセッサ設計者により供給された RTL を変更できない場合、このフィールドは RAZ です。

DBGPID4

DBGPID4 はデバッグレジスタ 1012、オフセット `0xFD0` です。フォーマットは次のとおりです。

ビット [31:8] 予約、RAZ

4KB カウント、ビット [7:4]

このフィールドは、すべての ARMv7 実装について RAZ です。

JEP 106 継続コード、ビット [3:0]

実装定義の JEP 106 継続コード。

ARM により設計されたデバイスでは、このフィールドは `0x4` です。

C10.8.10 デバッグコンポーネント識別レジスタ (DBGCID0 ～ DBGCID3)

デバッグコンポーネント識別レジスタは、プロセッサを ARM デバッグインタフェース v5 のコンポーネントとして識別します。詳細については、『*ARM デバッグインタフェース v5、アーキテクチャ仕様書*』を参照して下さい。

デバッグコンポーネント識別レジスタには、次の特徴があります。

- デバッグレジスタ 1020 ～ 1023、オフセット `0xFF0` ～ `0xFFC`
- 読み出し専用レジスタ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- occupy the last four words of the 4KB block of debug registers
- when the Security Extensions are implemented, are Common registers.

Table C10-24 lists the Debug Component Identification Registers.

Table C10-24 Debug Component Identification Registers

Register: Number	Offset	Description	Reference
1020	`0xFF0`	Debug Component ID0	*DBGCID0*
1021	`0xFF4`	Debug Component ID1	*DBGCID1* on page C10-104
1022	`0xFF8`	Debug Component ID2	*DBGCID2* on page C10-104
1023	`0xFFC`	Debug Component ID3	*DBGCID3* on page C10-104

Only bits [7:0] of each Debug Component ID Register are used. This means that the format of each register is:

31 … 8	7 … 0
Reserved, RAZ	Component ID data

The four Debug Component ID Registers can be considered as defining a single 32-bit Component ID, as shown in Figure C10-3. The value of this Component ID is fixed.

Figure C10-3 Mapping between Debug Component ID Registers and the 32-bit Component ID value

The following subsections describe the formats of each of the Debug Component ID Registers.

DBGCID0

DBGCID0 is debug register 1020 at offset `0xFF0`. Its format is:

Bits [31:8] Reserved, RAZ.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- デバッグレジスタの 4KB ブロックのうち、最後の 4 ワードを占有します。
- セキュリティ拡張機能が実装されている場合、共通レジスタです。

デバッグコンポーネント識別レジスタの一覧を、表 C10-24 に示します。

表 C10-24 デバッグコンポーネント識別レジスタ

レジスタ 番号	オフセット	説明	参照先
1020	`0xFF0`	デバッグコンポーネント識別レジスタ 0	*DBGCID0*
1021	`0xFF4`	デバッグコンポーネント識別レジスタ 1	*DBGCID1* P. C10-104
1022	`0xFF8`	デバッグコンポーネント識別レジスタ 2	*DBGCID2* P. C10-104
1023	`0xFFC`	デバッグコンポーネント識別レジスタ 3	*DBGCID3* P. C10-104

各デバッグコンポーネント識別レジスタについて、ビット [7:0] のみが使用されます。各レジスタのフォーマットは次のとおりです。

31 … 8	7 … 0
予約、RAZ	コンポーネント ID データ

4 つのデバッグコンポーネント識別レジスタは、図 C10-3 に示すように、単一の 32 ビットコンポーネント ID を定義するものと考えることができます。このコンポーネント ID の値は固定です。

図 C10-3 デバッグコンポーネント識別レジスタと 32 ビットのコンポーネント ID 値とのマッピング

以下のサブセクションでは、デバッグコンポーネント識別レジスタそれぞれのフォーマットについて説明します。

DBGCID0

DBGCID0 はデバッグレジスタ 1020、オフセット `0xFF0` です。フォーマットは次のとおりです。

ビット [31:8] 予約、RAZ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Preamble byte 0, bits [7:0]

This byte has the value `0x0D`.

DBGCID1

DBGCID1 is debug register 1021 at offset `0xFF4`. Its format is:

Bits [31:8] Reserved, RAZ.

Component class, bits [7:4]

This field has the value `0x9`, indicating an ARM Debug component.

Preamble, bits [3:0]

This field has the value `0x0`.

DBGCID2

DBGCID2 is debug register 1022 at offset `0xFF8`. Its format is:

Bits [31:8] Reserved, RAZ.

Preamble byte 2, bits [7:0]

This field has the value `0x05`.

DBGCID3

DBGCID3 is debug register 1023 at offset `0xFFC`. Its format is:

Bits [31:8] Reserved, RAZ.

Preamble byte 3, bits [7:0]

This field has the value `0xB1`.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

序文バイト 0、ビット [7:0]

このバイトの値は `0x0D` です。

DBGCID1

DBGCID1 はデバッグレジスタ 1021、オフセット `0xFF4` です。フォーマットは次のとおりです。

ビット [31:8] 予約、RAZ

コンポーネントクラス、ビット [7:4]

このフィールドの値は `0x9` で、ARM デバッグコンポーネントを示しています。

序文、ビット [3:0]

このフィールドの値は `0x0` です。

DBGCID2

DBGCID2 はデバッグレジスタ 1022、オフセット `0xFF8` です。フォーマットは次のとおりです。

ビット [31:8] 予約、RAZ

序文バイト 2、ビット [7:0]

このフィールドの値は `0x05` です。

DBGCID3

DBGCID3 はデバッグレジスタ 1023、オフセット `0xFFC` です。フォーマットは次のとおりです。

ビット [31:8] 予約、RAZ

序文バイト 3、ビット [7:0]

このフィールドの値は `0xB1` です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9 Performance monitor registers

v6 Debug and v6.1 Debug

These registers are not defined in v6 Debug and v6.1 Debug.

v7 Debug The performance monitors are an optional feature in v7 Debug. If implemented, they are registers in CP15 c9, see *CP15 c9 register map* on page C9-10.

The following subsections describe the performance monitor registers:

- *c9, Performance Monitor Control Register (PMCR)*
- *c9, Count Enable Set Register (PMCNTENSET)* on page C10-108
- *c9, Count Enable Clear Register (PMCNTENCLR)* on page C10-109
- *c9, Overflow Flag Status Register (PMOVSR)* on page C10-110
- *c9, Software Increment Register (PMSWINC)* on page C10-112
- *c9, Event Counter Selection Register (PMSELR)* on page C10-113
- *c9, Cycle Count Register (PMCCNTR)* on page C10-114
- *c9, Event Type Select Register (PMXEVTYPER)* on page C10-115
- *c9, Event Count Register (PMXEVCNTR)* on page C10-116
- *c9, User Enable Register (PMUSERENR)* on page C10-117
- *c9, Interrupt Enable Set Register (PMINTENSET)* on page C10-118
- *c9, Interrupt Enable Clear Register (PMINTENCLR)* on page C10-119.

C10.9.1 c9, Performance Monitor Control Register (PMCR)

The Performance Monitor Control Register, PMCR:

- provides details of the performance monitor implementation, including the number of counters implemented
- configures and controls the counters.

The PMCR:

- is a 32-bit read/write CP15 register, with more restricted access to some bits
- is accessible in:
  - — privileged modes
  - — User mode only when the PMUSERENR.EN bit is set to 1
- when the Security Extensions are implemented, is a Common register
- is accessed using an `MRC` or `MCR` command with `<CRn>` set to c9, `<opc1>` set to 0, `<CRm>` set to c12, and `<opc2>` set to 0
- has defined core logic reset values for its read/write bits.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9 パフォーマンスモニタ レジスタ

v6 デバッグと v6.1 デバッグ

これらのレジスタは、v6 デバッグおよび v6.1 デバッグでは定義されていません。

v7 デバッグ パフォーマンスモニタは、v7 デバッグのオプション機能です。実装されている場合、CP15 c9 のレジスタです。*CP15 c9 レジスタマップ* P. C9-10 を参照して下さい。

次のサブセクションでは、パフォーマンスモニタのレジスタについて説明します。

- *c9、パフォーマンスモニタ制御レジスタ (PMCR)*
- *c9、カウントイネーブルセット レジスタ (PMCNTENSET)* P. C10-108
- *c9、カウントイネーブルクリア レジスタ (PMCNTENCLR)* P. C10-109
- *c9、オーバフローフラグステータス レジスタ (PMOVSR)* P. C10-110
- *c9、ソフトウェアインクリメント レジスタ (PMSWINC)* P. C10-112
- *c9、イベントカウンタ選択レジスタ (PMSELR)* P. C10-113
- *c9、サイクルカウント レジスタ (PMCCNTR)* P. C10-114
- *c9、イベントタイプ選択レジスタ (PMXEVTYPER)* P. C10-115
- *c9、イベントカウント レジスタ (PMXEVCNTR)* P. C10-116
- *c9、ユーザイネーブル レジスタ (PMUSERENR)* P. C10-117
- *c9、割り込みイネーブルセット レジスタ (PMINTENSET)* P. C10-118
- *c9、割り込みイネーブルクリア レジスタ (PMINTENCLR)* P. C10-119

C10.9.1 c9、パフォーマンスモニタ制御レジスタ (PMCR)

パフォーマンスモニタ制御レジスタ (PMCR) は、次の目的に使用されます。

- 実装されているカウンタの数など、パフォーマンスモニタ実装の詳細を提供する。
- カウンタの構成と制御を行う。

PMCR には、次の特徴があります。

- 32 ビットの読み出し / 書き込み CP15 レジスタで、一部のビットはアクセスがさらに制限されている。
- 次のモードでアクセス可能である。
  - – 特権モード
  - – ユーザモードで、PMUSERENR.EN ビットが 1 にセットされている場合のみ
- セキュリティ拡張機能が実装されている場合、共通レジスタである。
- `MRC` または `MCR` コマンドで、`<CRn>` を c9、`<opc1>` を 0、`<CRm>` を c12、`<opc2>` を 0 に設定してアクセスされる。
- 読み出し / 書き込みビットについて、コアロジック リセット時の値が定義されている。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the PMCR is:

31 – 24	23 – 16	15 – 11	10 – 6	5	4	3	2	1	0
IMP	IDCODE	N	UNK/SBZP	DP	X	D	C	P	E

IMP, bits [31:24]

Implementer code. This is a read-only field with an IMPLEMENTATION DEFINED value.

The Implementer codes are allocated by ARM. Values have the same interpretation as bits [31:24] of the CP15 Main ID Register, see:

- *c0, Main ID Register (MIDR)* on page B3-81 for a VMSA implementation
- *c0, Main ID Register (MIDR)* on page B4-32 for a PMSA implementation.

IDCODE, bits [23:16]

Identification code. This is a read-only field with an IMPLEMENTATION DEFINED value.

Each implementer must maintain a list of identification codes that is specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

N, bits [15:11]

Number of event counters. This is a read-only field with an IMPLEMENTATION DEFINED value that indicates the number of counters implemented.

The value of this field is the number of counters implemented, from 0b00000 for no counters to 0b11111 for 31 counters.

An implementation can implement only the Clock Counter (PMCCNTR) Register. This is indicated by a value of 0b00000 for the N field.

Bits [10:6] Reserved, UNK/SBZP.

DP, bit [5] Disable PMCCNTR when prohibited. The possible values of this bit are:

0 Count is enabled in prohibited regions

1 Count is disabled in prohibited regions.

Prohibited regions are defined as regions where event counting would be prohibited. For example, if non-invasive debug is disabled in all Secure modes, the Secure state is a prohibited region. For details of non-invasive debug authentication see Chapter C7 *Non-invasive Debug Authentication*.

Note

This bit permits a Non-secure process to discard cycle counts that might be accumulated during periods when the other counts are prohibited because of security prohibitions. It is not a control to enhance security. The function of this bit is to avoid corruption of the count. See also *Interaction with Security Extensions* on page C9-7.

This is a read/write bit. Its core logic reset value is 0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

PMCR のフォーマットは次のとおりです。

31 – 24	23 – 16	15 – 11	10 – 6	5	4	3	2	1	0
IMP	IDCODE	N	UNK/SBZP	DP	X	D	C	P	E

IMP、ビット [31:24]

実装者コード。このフィールドは読み出し専用で、値は実装定義です。

実装者コードは ARM により割り当てられます。値の解釈は CP15 メイン ID レジスタのビット [31:24] と同じです。次のセクションを参照して下さい。

- *c0、メイン ID レジスタ (MIDR)* P. B3-81（VMSA 実装の場合）
- *c0、メイン ID レジスタ (MIDR)* P. B4-32 （PMSA 実装の場合）

IDCODE、ビット [23:16]

ID コード。このフィールドは読み出し専用で、値は実装定義です。

各実装で、実装者に固有の ID コード一覧を保持する必要があります。特定の実装は、実装者コードと ID コードの組み合わせにより識別されます。

N、ビット [15:11]

イベントカウンタの数。このフィールドは読み出し専用で、実装されているカウンタの数を示す実装定義の値です。

このフィールドの値は実装されているカウンタの数で、カウンタが存在しない場合の 0b00000 から、31 カウンタを示す 0b11111 までです。

実装で、クロックカウンタ (PMCCNTR) レジスタのみを実装することもできます。これは、N フィールドの値 0b00000 により示されます。

ビット [10:6] 予約、UNK/SBZP

DP、ビット [5] PMCCNTR が禁止されているときに非稼動にします。このビットの可能な値は次のとおりです。

0 禁止領域でカウントが稼動中です。

1 禁止領域でカウントが非稼動です。

禁止領域は、イベントのカウントが禁止されている領域と定義されます。例えば、すべてのセキュアモードで非侵襲性デバッグが不可能な場合、セキュア状態は禁止領域です。非侵襲性デバッグ認証の詳細については、第 C7 章*「非侵襲性デバッグの認証」*を参照して下さい。

注

このビットは、非セキュアプロセスがサイクルカウントを破棄することを許可するため、セキュリティ上の禁止により他のカウンタが禁止されている場合、一定期間にわたって破棄されたサイクルカウントが累積されることがあります。これはセキュリティを強化するための制御ではありません。このビットの機能は、カウントの破損を防止することです。*セキュリティ拡張機能との相互動作* P. C9-7 も参照して下さい。

読み出し / 書き込みビットです。コアロジック リセット時の値は 0 です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

X, bit [4] Export enable. The possible values of this bit are:

0 Export of events is disabled

1 Export of events is enabled.

This bit is used to permit events to be exported to another debug device, such as a trace macrocell, over an event bus. If the implementation does not include such an event bus, this bit is RAZ/WI.

This bit does not affect the generation of performance monitor interrupts, that can be implemented as a signal exported from the processor to an interrupt controller.

This is a read/write bit. Its core logic reset value is 0.

D, bit [3] Clock divider. The possible values of this bit are:

0 When enabled, PMCCNTR counts every clock cycle

1 When enabled, PMCCNTR counts once every 64 clock cycles.

This is a read/write bit. Its core logic reset value is 0.

C, bit [2] Clock counter reset. This is a write-only bit. The effects of writing to this bit are:

0 No action

1 Reset PMCCNTR to zero.

Note

Resetting PMCCNTR does not clear the PMCCNTR overflow flag to 0. For details see *c9, Overflow Flag Status Register (PMOVSR)* on page C10-110.

This bit is always RAZ.

P, bit [1] Event counter reset. This is a write-only bit. The effects of writing to this bit are:

0 No action

1 Reset all event counters, not including PMCCNTR, to zero.

Note

Resetting the event counters does not clear any overflow flags to 0. For details see *c9, Overflow Flag Status Register (PMOVSR)* on page C10-110.

This bit is always RAZ.

E, bit [0] Enable. The possible values of this bit are:

0 All counters, including PMCCNTR, are disabled

1 All counters are enabled.

Performance monitor overflow IRQs are only signaled when the enable bit is set to 1.

This is a read/write bit. Its core logic reset value is 0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

X、ビット[4] エクスポートイネーブル。このビットの可能な値は次のとおりです。

0 イベントのエクスポートが不可能です。

1 イベントのエクスポートが可能です。

このビットは、他のデバッグデバイス、例えばトレースマクロセルに対して、イベントバス経由でのイベントのエクスポートを許可するために使用されます。実装にこのようなイベントバスが含まれていない場合、このビットは RAZ/WI です。

このビットは、パフォーマンスモニタ割り込みの生成には影響しません。これらの割り込みは、プロセッサから割り込みコントローラへエクスポートされる信号として実装可能なためです。

読み出し / 書き込みビットです。コアロジック リセット時の値は 0 です。

D、ビット[3] クロックディバイダ。このビットの可能な値は次のとおりです。

0 稼動しているとき、PMCCNTR はすべてのクロックサイクルをカウントします。

1 稼動しているとき、PMCCNTR は 64 クロックサイクルごとに 1 回カウントされます。

読み出し / 書き込みビットです。コアロジック リセット時の値は 0 です。

C、ビット[2] クロックカウンタのリセット。このビットは書き込み専用です。このビットへの書き込みは、次の動作を行います。

0 動作を行いません。

1 PMCCNTR を 0 にリセットします。

注

PMCCNTR をリセットしても、PMCCNTR オーバフローフラグは 0 にクリアされません。詳細については、*c9、オーバフローフラグステータス レジスタ (PMOVSR)* P. C10-110 を参照して下さい。

このビットは常に RAZ です。

P、ビット[1] イベントカウンタのリセット。このビットは書き込み専用です。このビットへの書き込みは、次の動作を行います。

0 動作を行いません。

1 PMCCNTR を除くすべてのイベントカウンタを 0 にリセットします。

注

イベントカウンタをリセットしても、オーバフローフラグは 0 にクリアされません。詳細については、*c9、オーバフローフラグステータス レジスタ (PMOVSR)* P. C10-110 を参照して下さい。

このビットは常に RAZ です。

E、ビット[0] イネーブルビット。このビットの可能な値は次のとおりです。

0 PMCCNTR を含むすべてのカウンタが非稼動です。

1 すべてのカウンタが稼動中です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9.2 c9, Count Enable Set Register (PMCNTENSET)

The Count Enable Set Register, PMCNTENSET, is used to enable:

- the Cycle Count Register, PMCCNTR
- any implemented event counters, PMN*x*.

Reading the PMCNTENSET Register shows which counters are enabled. Counters are disabled using the Count Enable Clear Register, see *c9, Count Enable Clear Register (PMCNTENCLR)* on page C10-109.

The PMCNTENSET Register is:

- a 32-bit read/write CP15 register:
  - — reading the register shows which counters are enabled
  - — writing a 1 to a bit of the register enables the corresponding counter
  - — writing a 0 to a bit of the register has no effect
- accessible in:
  - — privileged modes
  - — User mode only when the PMUSERENR.EN bit is set to 1
- accessed using an `MRC` or `MCR` command with `<CRn>` set to c9, `<opc1>` set to 0, `<CRm>` set to c12, and `<opc2>` set to 1.

The format of the PMCNTENSET Register is:

31	30 … N	N-1 … 0
C	RAZ/WI	Event counter enable bits, P*x*, for *x* = 0 to (N-1)

Note

In the description of the PMCNTENSET Register:

- N is the number of event counters implemented, as defined by the PMCR.N field, see *c9, Performance Monitor Control Register (PMCR)* on page C10-105
- *x* refers to a single event counter, and takes values from 0 to (N-1).

C, bit [31] PMCCNTR enable bit.

See Table C10-25 on page C10-109 for the behavior of this bit on reads and writes.

Bits [30:N] RAZ/WI.

P*x*, bit [*x*], for *x* = 0 to (N-1)

Event counter *x*, PMN*x*, enable bit.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

パフォーマンスモニタ オーバフローの IRQ は、このイネーブルビットが 1 にセットされているときのみ通知されます。

このビットは読み出し / 書き込みです。コアロジック リセット時の値は 0 です。

C10.9.2 c9、カウントイネーブルセット レジスタ (PMCNTENSET)

カウントイネーブルセット レジスタ (PMCNTENSET) は、次のカウンタを稼動するために使用されます。

- サイクルカウント レジスタ (PMCCNTR)
- 実装されている任意のイベントカウンタ (PMN*x*)

PMCNTENSET レジスタを読み出すと、どのカウンタが稼動中かを判定できます。カウンタを非稼動にするには、カウントイネーブルクリア レジスタを使用します。*c9、カウントイネーブルクリア レジスタ (PMCNTENCLR)* P. C10-109 を参照して下さい。

PMCNTENSET レジスタには、次の特徴があります。

- 32 ビットの読み出し / 書き込み CP15 レジスタで、次の特徴があります。
  - – このレジスタを読み出すと、どのカウンタが稼動中かを判定できます。
  - – レジスタのビットに 1 を書き込むと、対応するカウンタが稼動します。
  - – レジスタのビットに 0 を書き込んでも無効です。
- 次のモードでアクセス可能です。
  - – 特権モード
  - – ユーザモードで、PMUSERENR.EN ビットが 1 にセットされている場合のみ
- `MRC` または `MCR` コマンドで、`<CRn>` を c9、`<opc1>` を 0、`<CRm>` を c12、`<opc2>` を 1 に設定してアクセスされます。

PMCNTENSET レジスタのフォーマットは次のとおりです。

31	30 … N	N – 1 … 0
C	RAZ/WI	イベントカウンタイネーブル ビット、P*x*、*x* は 0 ～ (N – 1)

注

PMCNTENSET レジスタの説明に含まれている記号の意味は次のとおりです。

- N は、PMCR.N フィールドによって定義される、実装されているイベントカウンタの数です。*c9、パフォーマンスモニタ制御レジスタ (PMCR)* P. C10-105 を参照して下さい。
- *x* は単一のイベントカウンタを指し、値は 0 ～ (N–1) です。

C、ビット [31] PMCCNTR イネーブルビット。

このビットの読み出しおよび書き込みの動作については、P. C10-109 の表 C10-25 を参照して下さい。

ビット [30:N] RAZ/WI

P*x*、ビット [*x*]、*x* は 0 ～ (N – 1)

イベントカウンタ *x*(PMN*x*) のイネーブルビット

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996–1998, 2000, 2004–2008 ARM Limited. All rights reserved.

Table C10-25 shows the behavior of this bit on reads and writes.

Table C10-25 Read and write bit values for the PMCNTENSET Register

Value	Meaning on read	Action on write
0	Counter disabled	No action, write is ignored
1	Counter enabled	Enable counter

The contents of the PMCNTENSET Register are UNKNOWN on a core logic reset.

C10.9.3 c9, Count Enable Clear Register (PMCNTENCLR)

The Count Enable Clear Register, PMCNTENCLR, is used to disable:

- the Cycle Count Register, PMCCNTR
- any implemented event counters, PMN*x*.

Reading the PMCNTENCLR Register shows which counters are enabled. Counters are enabled using the Count Enable Set Register, see *c9, Count Enable Set Register (PMCNTENSET)* on page C10-108.

The PMCNTENCLR Register is:

- a 32-bit read/write CP15 register:
  - — reading the register shows which counters are enabled
  - — writing a 1 to a bit of the register disables the corresponding counter
  - — writing a 0 to a bit of the register has no effect
- accessible in:
  - — privileged modes
  - — User mode only when the PMUSERENR.EN bit is set to 1
- when the Security Extensions are implemented, a Common register
- accessed using an `MRC` or `MCR` command with `<CRn>` set to c9, `<opc1>` set to 0, `<CRm>` set to c12, and `<opc2>` set to 2.

The format of the PMCNTENCLR Register is:

31	30 … N	N-1 … 0
C	RAZ/WI	Event counter disable bits, P*x*, for *x* = 0 to (N-1)

Note

In the description of the PMCNTENCLR Register, N and *x* have the meanings used in the description of the PMCNTENSET Register, see *c9, Count Enable Set Register (PMCNTENSET)* on page C10-108.

C, bit [31] PMCCNTR disable bit.

See Table C10-26 on page C10-110 for the behavior of this bit on reads and writes.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

このビットの読み出しおよび書き込みの動作を、表 C10-25 に示します。

表 C10-25 PMCNTENSET レジスタのビット値の読み出しおよび書き込み

値	読み出しの意味	書き込み時の動作
0	カウンタが非稼動です。	動作は行われません。書き込みは無視されます。
1	カウンタが稼動中です。	カウンタを稼動します。

コアロジック リセット時の PMCNTENSET レジスタの内容は未知です。

C10.9.3 c9、カウントイネーブルクリア レジスタ (PMCNTENCLR)

カウントイネーブルクリア レジスタ (PMCNTENCLR) は、次のカウンタを非稼動にするために使用されます。

- サイクルカウント レジスタ (PMCCNTR)
- 実装されている任意のイベントカウンタ (PMN*x*)

PMCNTENCLR レジスタを読み出すと、どのカウンタが稼動中かを判定できます。カウンタを稼動するには、カウントイネーブルセット レジスタを使用します。*c9、カウントイネーブルセット レジスタ (PMCNTENSET)* P. C10-108 を参照して下さい。

PMCNTENCLR レジスタには、次の特徴があります。

- 32 ビットの読み出し / 書き込み CP15 レジスタで、次の特徴があります。
  - – このレジスタを読み出すと、どのカウンタが稼動中かを判定できます。
  - – レジスタのビットに 1 を書き込むと、対応するカウンタが非稼動になります。
  - – レジスタのビットに 0 を書き込んでも無効です。
- 次のモードでアクセス可能です。
  - – 特権モード
  - – ユーザモードで、PMUSERENR.EN ビットが 1 にセットされている場合のみ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。
- `MRC` または `MCR` コマンドで、`<CRn>` を c9、`<opc1>` を 0、`<CRm>` を c12、`<opc2>` を 2 に設定してアクセスされます。

PMCNTENCLR レジスタのフォーマットは次のとおりです。

31	30 … N	N − 1 … 0
C	RAZ/WI	イベントカウンタディセーブル ビット、P*x*、*x* は 0 ～ (N − 1)

注

PMCNTENCLR レジスタの説明で、N と *x* は、PMCNTENSET レジスタの説明で使用されているのと同じ意味です。*c9、カウントイネーブルセット レジスタ (PMCNTENSET)* P. C10-108 を参照して下さい。

C、ビット [31] PMCCNTR ディセーブルビット。

このビットの読み出しおよび書き込みの動作については、P. C10-110 の表 C10-26 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Bits [30:N] RAZ/WI.

P*x*, bit [*x*], for *x* = 0 to (N-1)

Event counter *x*, PMN*x*, disable bit.

Table C10-26 shows the behavior of this bit on reads and writes.

Table C10-26 Read and write bit values for the PMCNTENCLR Register

Value	Meaning on read	Action on write
0	Counter disabled	No action, write is ignored
1	Counter enabled	Disable counter

The contents of the PMCNTENCLR Register are UNKNOWN on a core logic reset.

Note

The PMCR.E Enable bit can be used to override the settings in this register and disable all counters including PMCCNTR, see *c9, Performance Monitor Control Register (PMCR)* on page C10-105. The counter enable register retains its value when the Enable bit is 0, even though its settings are ignored.

C10.9.4 c9, Overflow Flag Status Register (PMOVSR)

The Overflow Flag Status Register, PMOVSR, holds the state of the overflow flags for:

- the Cycle Count Register, PMCCNTR
- each of the implemented event counters, PMN*x*.

To clear those flags you must write to the PMOVSR.

The PMOVSR is:

- a 32-bit read/write CP15 register:
  - — reading the register shows the state of the overflow flags
  - — writing a 1 to a bit of the register clears the corresponding flag
  - — writing a 0 to a bit of the register has no effect
- accessible in:
  - — privileged modes
  - — User mode only when the PMUSERENR.EN bit is set to 1
- when the Security Extensions are implemented, a Common register
- accessed using an MRC or MCR command with <CRn> set to c9, <opc1> set to 0, <CRm> set to c12, and <opc2> set to 3.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ビット[30:N] RAZ/WI

P*x*、ビット [*x*]、*x* は 0 ～ (N − 1)

イベントカウンタ *x*(PMN*x*) のディセーブルビット。

このビットの読み出しおよび書き込みの動作を、表 C10-26 に示します。

表 C10-26 PMCNTENCLR レジスタのビット値の読み出しおよび書き込み

値	読み出しの意味	書き込み時の動作
0	カウンタが非稼動です。	動作は行われません。書き込みは無視されます。
1	カウンタが稼動中です。	カウンタを非稼動にします。

コアロジック リセット時の PMCNTENCLR レジスタの内容は未知です。

注

PMCR.E イネーブルビットを使用して、このレジスタの設定をオーバライドし、PMCCNTR を含むすべてのカウンタを非稼動にできます。*c9、パフォーマンスモニタ制御レジスタ (PMCR)* P. C10-105 を参照して下さい。このイネーブルビットが 0 のとき、カウンタイネーブル レジスタの値は保持されますが、その設定は無視されます。

C10.9.4 c9、オーバフローフラグステータス レジスタ (PMOVSR)

オーバフローフラグステータス レジスタ (PMOVSR) は、次のカウンタについてオーバフローフラグの状態を保持します。

- サイクルカウント レジスタ (PMCCNTR)
- 実装されている各イベントカウンタ (PMN*x*)

これらのフラグをクリアするには、PMOVSR に書き込む必要があります。

PMOVSR には、次の特徴があります。

- 32 ビットの読み出し / 書き込み CP15 レジスタで、次の特徴があります。
  - ー レジスタを読み出すと、オーバフローフラグの状態を判定できます。
  - ー レジスタのビットに 1 を書き込むと、対応するフラグがクリアされます。
  - ー レジスタのビットに 0 を書き込んでも無効です。
- 次のモードでアクセス可能です。
  - ー 特権モード
  - ー ユーザモードで、PMUSERENR.EN ビットが 1 にセットされている場合のみ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。
- MRC または MCR コマンドで、<CRn> を c9、<opc1> を 0、<CRm> を c12、<opc2> を 3 に設定してアクセスされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the PMOVSR is:

31	30 … N	N-1 … 0
C	RAZ/WI	Event counter overflow flags, P*x*, for *x* = 0 to (N-1)

Note

In the description of the PMOVSR, N and *x* have the meanings used in the description of the PMCNTENSET Register, see *c9, Count Enable Set Register (PMCNTENSET)* on page C10-108.

C, bit [31] PMCCNTR overflow flag.

Table C10-27 shows the behavior of this bit on reads and writes.

Bits [30:N] RAZ/WI.

P*x*, bit [*x*], for *x* = 0 to (N-1)

Event counter *x*, PMN*x*, overflow flag.

Table C10-27 shows the behavior of this bit on reads and writes.

Table C10-27 Read and write bit values for the PMOVSR

Value	Meaning on read	Action on write
0	Counter has not overflowed	No action, write is ignored
1	Counter has overflowed	Clear flag to 0

The contents of the PMOVSR are UNKNOWN on a core logic reset.

Note

The overflow flag values for individual counters are retained until cleared to 0 by a write to the PMOVSR or processor reset, even if the counter is later disabled by writing to the PMCNTENCLR register or through the PMCR.E Enable bit. The overflow flags are also not cleared to 0 when the counters are reset through the Event counter reset or Clock counter reset bits in the PMCR.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

PMOVSR のフォーマットは次のとおりです。

31	30 … N	N − 1 … 0
C	RAZ/WI	イベントカウンタオーバフロー フラグ、P*x*、*x* は 0 ～ (N − 1)

注

PMOVSR レジスタの説明で、N と *x* は、PMCNTENSET レジスタの説明で使用されているのと同じ意味です。*c9、カウントイネーブルセット レジスタ (PMCNTENSET)* P. C10- 108 を参照して下さい。

C、ビット [31] PMCCNTR オーバフローフラグ。

このビットの読み出しおよび書き込みの動作を、表 C10- 27 に示します。

ビット [30:N] RAZ/WI

P*x*、ビット [*x*]、*x* は 0 ～ (N − 1)

イベントカウンタ *x*(PMN*x*) のオーバフローフラグ。

このビットの読み出しおよび書き込みの動作を、表 C10- 27 に示します。

表 C10- 27 PMOVSR のビット値の読み出しおよび書き込み

値	読み出しの意味	書き込み時の動作
0	カウンタはオーバフローしていません。	動作は行われません。書き込みは無視されます。
1	カウンタはオーバフローしました。	フラグを 0 にクリアします。

コアロジック リセット時の PMOVSR の内容は未知です。

注

各カウンタのオーバフローフラグの値は、PMOVSR への書き込みまたはプロセッサのリセットによって 0 にクリアされるまで保持されます。カウンタが後で PMCNTENCLR レジスタへの書き込みや PMCR.E イネーブルビットによって非稼動になったとしても、値は変化しません。また、カウンタがイベントカウンタ リセットや、PMCR のクロックカウンタリセット ビットによってリセットされても、オーバフローフラグは 0 にクリアされません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9.5 c9, Software Increment Register (PMSWINC)

The Software Increment Register, PMSWINC, increments a counter that is configured to count the Software count event, event `0x00`.

The PMSWINC Register is:

- a 32-bit write-only CP15 register
- accessible in:
  - — privileged modes
  - — User mode only when the PMUSERENR.EN bit is set to 1
- when the Security Extensions are implemented, a Common register
- accessed using an `MCR` command with `<CRn>` set to c9, `<opc1>` set to 0, `<CRm>` set to c12, and `<opc2>` set to 4.

The format of the PMSWINC Register is:

31 … N	N-1 … 0
SBZ	Event counter software increment bits, P*x*, for *x* = 0 to (N-1)

Note

In the description of the PMSWINC Register, N and *x* have the meanings used in the description of the PMCNTENSET Register, see *c9, Count Enable Set Register (PMCNTENSET)* on page C10-108.

Bits [31:N] Reserved, SBZ.

P*x*, bit [*x*], for *x* = 0 to (N-1)

Event counter *x*, PMN*x*, software increment bit. This is a write-only bit. The effects of writing to this bit are:

0 No action, the write is ignored

1, if PMN*x* is configured to count the Software count event

Increment PMN*x* Register by 1

1, if PMN*x* is not configured to count the Software count event

UNPREDICTABLE.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C10.9.5 c9、ソフトウェアインクリメント レジスタ (PMSWINC)

ソフトウェアインクリメント レジスタ (PMSWINC) は、ソフトウェアカウント イベント（イベント `0x00`）をカウントするように構成されているカウンタをインクリメントします。

PMSWINC レジスタには、次の特徴があります。

- 32 ビットの書き込み専用 CP15 レジスタ
- 次のモードでアクセス可能である。
  - – 特権モード
  - – ユーザモードで、PMUSERENR.EN ビットが 1 にセットされている場合のみ
- セキュリティ拡張機能が実装されている場合、共通レジスタです。
- `MCR` コマンドで、`<CRn>` を c9、`<opc1>` を 0、`<CRm>` を c12、`<opc2>` を 4 に設定してアクセスされます。

PMSWINC レジスタのフォーマットは次のとおりです。

31 … N	N − 1 … 0
SBZ	イベントカウンタ ソフトウェアインクリメント ビット、P*x*、*x* は 0 ～ (N − 1)

注

PMSWINC レジスタの説明で、N と *x* は、PMCNTENSET レジスタの説明で使用されているのと同じ意味です。*c9、カウント イネーブルセット レジスタ (PMCNTENSET)* P. C10-108 を参照して下さい。

ビット [31:N] 予約、SBZ

P*x*、ビット [*x*]、*x* は 0 ～ (N − 1)

イベントカウンタ *x*(PMN*x*) のソフトウェアインクリメント ビット。このビットは書き込み専用です。このビットへの書き込みは、次の動作を行います。

0 動作は行われません。書き込みは無視されます。

1（PMN*x* がソフトウェアカウント イベントをカウントするように構成されている場合）

PMN*x* レジスタを 1 インクリメントします。

1（PMN*x* がソフトウェアカウント イベントをカウントするように構成されていない場合）

予測不能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9.6 c9, Event Counter Selection Register (PMSELR)

The Event Counter Selection Register, PMSELR, selects the current event counter, PMN*x*. When a particular event counter is selected:

- the PMXEVTYPER can be used to set the event that increments that counter, or to read the current configuration, see *c9, Event Type Select Register (PMXEVTYPER)* on page C10-115
- the PMXEVCNTR can be used to read the current value of that counter, or to write a value to that counter, see *c9, Event Count Register (PMXEVCNTR)* on page C10-116

The PMSELR is:

- a 32-bit read/write CP15 register
- accessible in:
  - — privileged modes
  - — User mode only when the PMUSERENR.EN bit is set to 1
- when the Security Extensions are implemented, a Common register
- accessed using an `MRC` or `MCR` command with `<CRn>` set to c9, `<opc1>` set to 0, `<CRm>` set to c12, and `<opc2>` set to 5.

The format of the PMSELR is:

31 … 5	4 … 0
Reserved, UNK/SBZP	SEL

Bits [31:5] Reserved, UNK/SBZP.

SEL, bits [4:0]

Selection value of the current event counter, PMN*x*, where *x* is the value held in this field. This field can take any value from 0 (0b00000) to 30 (0b11110).

If this field is set to a value greater than or equal to the number of implemented counters the results are UNPREDICTABLE. The number of implemented counters is defined by the PMCR.N field, see *c9, Performance Monitor Control Register (PMCR)* on page C10-105.

The value of 0b11111 is Reserved and must not be used.

The contents of the PMSELR are UNKNOWN on a core logic reset.

The SEL field identifies which event counter, PMN_{SEL}, is accessed by PMXEVTYPER and PMXEVCNTR, see *c9, Event Type Select Register (PMXEVTYPER)* on page C10-115 and *c9, Event Count Register (PMXEVCNTR)* on page C10-116.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9.6 c9、イベントカウンタ選択レジスタ (PMSELR)

イベントカウンタ選択レジスタ (PMSELR) は、現在のイベントカウンタ PMN*x* を選択します。特定のイベントカウンタが選択されると、次の動作が可能になります。

- PMXEVTYPER を使用して、そのカウンタをインクリメントするイベントを設定するか、現在の構成を読み出すことができます。*c9、イベントタイプ選択レジスタ (PMXEVTYPER)* P. C10-115 を参照して下さい。
- PMXEVCNTR を使用して、カウンタの現在の値を読み出すか、カウンタに値を書き込むことができます。*c9、イベントカウント レジスタ (PMXEVCNTR)* P. C10-116 を参照して下さい。

PMSELR には、次の特徴があります。

- 32 ビットの読み出し / 書き込み CP15 レジスタ
- 次のモードでアクセス可能である。
  - – 特権モード
  - – ユーザモードで、PMUSERENR.EN ビットが 1 にセットされている場合のみ
- セキュリティ拡張機能が実装されている場合、共通レジスタである。
- `MRC` または `MCR` コマンドで、`<CRn>` を c9、`<opc1>` を 0、`<CRm>` を c12、`<opc2>` を 5 に設定してアクセスされる。

PMSELR のフォーマットは次のとおりです。

31 … 5	4 … 0
予約、UNK/SBZP	SEL

ビット[31:5] 予約、UNK/SBZP

SEL、ビット [4:0]

現在のイベントカウンタを選択する値、PMN*x*。ここで *x* は、このフィールドに保持されている値です。このフィールドには、0 (0b00000) ～ 30 (0b11110) の任意の値を指定できます。

このフィールドに設定されている値が、実装されているカウンタ数と等しい、またはより大きい場合、結果は予測不能です。実装されているカウンタ数は、PMCR.N フィールドにより定義されます。*c9、パフォーマンスモニタ制御レジスタ (PMCR)* P. C10-105 を参照して下さい。

値 0b11111 は予約されており、使用できません。

コアロジック リセット時の PMSELR の内容は未知です。

SEL フィールドは、どのイベントカウンタ（PMN_{SEL}）が PMXEVTYPER および PMXEVCNTR によりアクセスされるかを識別します。*c9、イベントタイプ選択レジスタ (PMXEVTYPER)* P. C10-115 および *c9、イベントカウント レジスタ (PMXEVCNTR)* P. C10-116 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9.7 c9, Cycle Count Register (PMCCNTR)

The Cycle Count Register, PMCCNTR, counts processor clock cycles. Depending on the value of the PMCR.D bit, PMCCNTR increments either on every processor clock cycle or on every 64th processor clock cycle. See *c9, Performance Monitor Control Register (PMCR)* on page C10-105.

The PMCCNTR is:

- a 32-bit read/write CP15 register
- accessible in:
  - — privileged modes
  - — User mode only when the PMUSERENR.EN bit is set to 1
- when the Security Extensions are implemented, a Common register
- accessed using an `MRC` or `MCR` command with `<CRn>` set to c9, `<opc1>` set to 0, `<CRm>` set to c13, and `<opc2>` set to 0.

The format of the PMCCNTR is:

31 … 0
CCNT

CCNT, bits [31:0]

Cycle count. Depending on the value of the PMCR.D bit, this field increments either:

- every processor clock cycle
- every 64th processor clock cycle.

The contents of the PMCCNTR are UNKNOWN on a core logic reset.

The PMCCNTR.CCNT value can be reset to zero by writing a 1 to the PMCR.C bit, see *c9, Performance Monitor Control Register (PMCR)* on page C10-105.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9.7 c9、サイクルカウント レジスタ (PMCCNTR)

サイクルカウント レジスタ (PMCCNTR) は、プロセッサのクロックサイクルをカウントします。PMCR.D ビットの値に応じて、PMCCNTR はプロセッサのすべてのクロックサイクルについて、または 64 クロックサイクルごとにインクリメントされます。*c9、パフォーマンスモニタ制御レジスタ (PMCR)* P. C10- 105 を参照して下さい。

PMCCNTR には、次の特徴があります。

- 32 ビットの読み出し / 書き込み CP15 レジスタ
- 次のモードでアクセス可能である。
  - ー 特権モード
  - ー ユーザモードで、PMUSERENR.EN ビットが 1 にセットされている場合のみ
- セキュリティ拡張機能が実装されている場合、共通レジスタである。
- `MRC` または `MCR` コマンドで、`<CRn>` を c9、`<opc1>` を 0、`<CRm>` を c13、`<opc2>` を 0 に設定してアクセスされる。

PMCCNTR のフォーマットは次のとおりです。

31 … 0
CCNT

CCNT、ビット [31:0]

サイクルカウント。PMCR.D ビットの値に応じて、このフィールドは次のいずれかでインクリメントされます。

- プロセッサのすべてのクロックサイクル
- プロセッサの 64 クロックサイクルごと

コアロジック リセット時の PMCCNTR の内容は未知です。

PMCCNTR.CCNT の値は、PMCR.C ビットに 1 を書き込むことで 0 にリセットできます。*c9、パフォーマンスモニタ制御レジスタ (PMCR)* P. C10- 105 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9.8 c9, Event Type Select Register (PMXEVTYPER)

The Event Type Select Register, PMXEVTYPER, configures which event increments the current event counter, PMN*x*, or to read the current configuration. PMSELR selects the current event counter, see *c9, Event Counter Selection Register (PMSELR)* on page C10-113.

The PMXEVTYPER is:

- a 32-bit read/write CP15 register
- accessible in:
  - — privileged modes
  - — User mode only when the PMUSERENR.EN bit is set to 1
- when the Security Extensions are implemented, a Common register
- accessed using an `MRC` or `MCR` command with `<CRn>` set to c9, `<opc1>` set to 0, `<CRm>` set to c13, and `<opc2>` set to 1.

The format of the PMXEVTYPER is:

31 … 8	7 … 0
Reserved, UNK/SBZP	evtCount

Bits [31:8] Reserved, UNK/SBZP.

evtCount, bits [7:0]

Event to count. The Event number of the event that is counted by the current event counter, PMN*x*. For more information, see *Event numbers*.

The contents of each of the Event Type Select Registers are UNKNOWN on a core logic reset.

Event numbers

Event numbers are used in the PMXEVTYPER to determine the event that causes an event counter to increment. These event numbers are split into two ranges:

`0x00-0x3F` Common features. Reserved for the specified events. When an ARMv7 processor supports monitoring of an event that is assigned a number in this range, if possible it must use that number for the event. Unassigned values are reserved and might be used to define additional common events in future editions of this manual.

`0x40-0xFF` IMPLEMENTATION DEFINED features.

For more information, including the assigned values in the common features range, see *Event numbers* on page C9-13.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9.8 c9、イベントタイプ選択レジスタ (PMXEVTYPER)

イベントタイプ選択レジスタ (PMXEVTYPER) は、現在のイベントカウンタ (PMN*x*) をインクリメントするイベントを構成するか、現在の構成を読み出すために使用されます。現在のイベントカウンタは、PMSELR によって選択されます。*c9、イベントカウンタ選択レジスタ (PMSELR)* P. C10- 113 を参照して下さい。

PMXEVTYPER には、次の特徴があります。

- 32 ビットの読み出し / 書き込み CP15 レジスタ
- 次のモードでアクセス可能である。
  - – 特権モード
  - – ユーザモードで、PMUSERENR.EN ビットが 1 にセットされている場合のみ
- セキュリティ拡張機能が実装されている場合、共通レジスタである。
- `MRC` または `MCR` コマンドで、`<CRn>` を c9、`<opc1>` を 0、`<CRm>` を c13、`<opc2>` を 1 に設定してアクセスされる。

PMXEVTYPER のフォーマットは次のとおりです。

31 … 8	7 … 0
予約、UNK/SBZP	evtCount

ビット[31:8] 予約、UNK/SBZP

evtCount、ビット [7:0]

カウントするイベント。現在のイベントカウンタ (PMN*x*) によってカウントされるイベントのイベント番号。詳細については、*イベント番号*を参照して下さい。

コアロジック リセット時の各イベントタイプ選択レジスタの内容は未知です。

イベント番号

イベント番号は、イベントカウンタのインクリメントを引き起こすイベントを、PMXEVTYPER が特定するために使用されます。これらのイベント番号は、2 つの範囲に分けられます。

`0x00` 〜 `0x3F` 共通機能。指定のイベント用に予約されています。ARMv7 プロセッサが、この範囲の番号を割り当てられているイベントの監視をサポートする場合、可能な限りそのイベントについてこの番号を使用する必要があります。未割り当ての値は予約されており、本書の将来のエディションで、追加の共通イベントを定義するために使用される可能性があります。

`0x40` 〜 `0xFF` 実装定義の機能

共通機能の範囲に割り当てられている値など、詳細については*イベント番号* P. C9- 13 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9.9 c9, Event Count Register (PMXEVCNTR)

The Event Count Register, PMXEVCNTR, is used to read or write the value of the current event counter, PMN*x*. PMSELR selects the current event counter, see *c9, Event Counter Selection Register (PMSELR)* on page C10-113.

The PMXEVCNTR is:

- a 32-bit read/write CP15 register
- accessible in:
  - — privileged modes
  - — User mode only when the PMUSERENR.EN bit is set to 1
- when the Security Extensions are implemented, a Common register
- accessed using an `MRC` or `MCR` command with `<CRn>` set to c9, `<opc1>` set to 0, `<CRm>` set to c13, and `<opc2>` set to 2.

The format of the PMXEVCNTR is:

31	0
PMNX	

PMNX, bits [31:0]

Value of the current event counter, PMN*x*.

A read of the PMXEVCNTR always returns the current value of the register.

The contents of each of the Event Count Registers are UNKNOWN on a core logic reset.

Note

You can write to the PMXEVCNTR even when the counter is disabled. This is true regardless of why the counter is disabled, which can be any of:

- because 1 has been written to the appropriate bit in the PMCNTENCLR
- because the PMCR.E bit is set to 0
- by the non-invasive debug authentication.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C10.9.9 c9、イベントカウント レジスタ (PMXEVCNTR)

イベントカウント レジスタ (PMXEVCNTR) は、現在のイベントカウンタ (PMN*x*) の値を読み出しまたは書き込みするために使用されます。現在のイベントカウンタは、PMSELR によって選択されます。*c9、イベントカウンタ選択レジスタ (PMSELR)* P. C10- 113 を参照して下さい。

PMXEVCNTR には、次の特徴があります。

- 32 ビットの読み出し / 書き込み CP15 レジスタ
- 次のモードでアクセス可能です。
  - ー 特権モード
  - ー ユーザモードで、PMUSERENR.EN ビットが 1 にセットされている場合のみ
- セキュリティ拡張機能が実装されている場合、共通レジスタである。
- `MRC` または `MCR` コマンドで、`<CRn>` を c9、`<opc1>` を 0、`<CRm>` を c13、`<opc2>` を 2 に設定してアクセスされる。

PMXEVCNTR のフォーマットは次のとおりです。

31	0
PMNX	

PMNX、ビット [31:0]

現在のイベントカウンタ (PMN*x*) の値。

PMXEVCNTR を読み出すと、レジスタの現在の値が常に返されます。

コアロジック リセット時の各イベントカウント レジスタの内容は未知です。

注

PMXEVCNTR には、カウンタが非稼動のときも書き込みを行うことができます。これは、カウンタが次のいずれの理由で非稼動になっているかに関係しません。

- PMCNTENCLR の対応するビットに 1 が書き込まれたため
- PMCR.E ビットが 0 にセットされているため
- 非侵襲性デバッグ認証によって

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9.10 c9, User Enable Register (PMUSERENR)

The User Enable Register, PMUSERENR, enables or disables User mode access to the performance monitors.

The PMUSERENR is:

- a 32-bit read/write CP15 register, with access that depends on the current mode:
  - — in a privileged mode, the PMUSERENR is a read/write register
  - — in User mode, the PMUSERENR is a read-only register
- when the Security Extensions are implemented, a Common register
- accessed using an MRC or MCR command with <CRn> set to c9, <opc1> set to 0, <CRm> set to c14, and <opc2> set to 0.

The format of the PMUSERENR is:

31 … 1	0
Reserved, UNK/SBZP	EN

Bits [31:1] Reserved, UNK/SBZP.

EN, bit [0] User mode access enable bit. The possible values of this bit are:

0 User mode access to performance monitors disabled

1 User mode access to performance monitors enabled.

Some MCR and MRC instructions used to access the performance monitors are UNDEFINED in User mode when User mode access to the performance monitors is disabled. For more information, see *Access permissions* on page C9-12.

The PMUSERENR.EN bit is set to 0 on a core logic reset.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9.10 c9、ユーザイネーブル レジスタ (PMUSERENR)

ユーザイネーブル レジスタ (PMUSERENR) は、パフォーマンスモニタへのユーザモードでのアクセスを可能または不可能にするために使用されます。

PMUSERENR には、次の特徴があります。

- 32 ビットの読み出し / 書き込み CP15 レジスタで、アクセス権は現在のモードに依存する。
  - – 特権モードでは、PMUSERENR は読み出し / 書き込みレジスタです。
  - – ユーザモードでは、PMUSERENR は読み出し専用レジスタです。
- セキュリティ拡張機能が実装されている場合、共通レジスタである。
- MRC または MCR コマンドで、<CRn> を c9、<opc1> を 0、<CRm> を c14、<opc2> を 0 に設定してアクセスされる。

PMUSERENR のフォーマットは次のとおりです。

31 … 1	0
予約、UNK/SBZP	EN

ビット[31:1] 予約、UNK/SBZP

EN、ビット[0] ユーザモードアクセス イネーブルビット。このビットの可能な値は次のとおりです。

0 パフォーマンスモニタへのユーザモード アクセスは不可能です。

1 パフォーマンスモニタへのユーザモード アクセスが可能です。

パフォーマンスモニタにアクセスするため使用される MCR および MRC 命令の一部は、パフォーマンスモニタへのユーザモード アクセスが不可能なとき、ユーザモードでは未定義です。詳細については、*アクセス許可* P. C9- 12 を参照して下さい。

PMUSERENR.EN ビットは、コアロジック リセット時に 0 にセットされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C10.9.11 c9, Interrupt Enable Set Register (PMINTENSET)

The Interrupt Enable Set Register, PMINTENSET, enables the generation of interrupt requests on overflows from:

- the Cycle Count Register, PMCCNTR
- each implemented event counter, PMN*x*.

Reading the PMINTENSET Register shows which overflow interrupts are enabled. Counter overflow interrupts must be disabled using the PMINTENCLR Register, see *c9, Interrupt Enable Clear Register (PMINTENCLR)* on page C10-119.

The PMINTENSET Register is:

- A 32-bit read/write CP15 register:
  - — reading the register shows which overflow interrupts are enabled
  - — writing a 1 to a bit of the register enables the corresponding overflow interrupt
  - — writing a 0 to a bit of the register has no effect.
- Accessible only in privileged modes.

  The instructions that access the PMINTENSET Register are always UNDEFINED in User mode, even if the PMUSERENR.EN flag is set to 1, see *c9, User Enable Register (PMUSERENR)* on page C10-117.
- When the Security Extensions are implemented, a Common register.
- Accessed using an MRC or MCR command with <CRn> set to c9, <opc1> set to 0, <CRm> set to c14, and <opc2> set to 1.

The format of the PMINTENSET Register is:

31	30 … N	N-1 … 0
C	RAZ/WI	Event counter overflow interrupt enable bits, P*x*, for *x* = 0 to (N-1)

Note

In the description of the PMINTENSET Register, N and *x* have the meanings used in the description of the PMCNTENSET Register, see *c9, Count Enable Set Register (PMCNTENSET)* on page C10-108.

C, bit [31] PMCCNTR overflow interrupt enable bit.

See Table C10-28 on page C10-119 for the behavior of this bit on reads and writes.

Bits [30:N] RAZ/WI.

P*x*, bit [*x*], for *x* = 0 to (N-1)

Event counter *x*, PMN*x*, overflow interrupt enable bit.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

C10.9.11 c9、割り込みイネーブルセット レジスタ (PMINTENSET)

割り込みイネーブルセット レジスタ (PMINTENSET) は、次のカウンタのオーバフローによる割り込み要求の生成を可能にします。

- サイクルカウント レジスタ (PMCCNTR)
- 実装されている各イベントカウンタ (PMN*x*)

PMINTENSET レジスタを読み出すと、どのオーバフロー割り込みが可能かを判定できます。カウンタオーバフロー割り込みを不可能にするには、PMINTENCLR レジスタを使用する必要があります。*c9、割り込みイネーブルクリア レジスタ (PMINTENCLR)* P. C10-119 を参照して下さい。

PMINTENSET レジスタには、次の特徴があります。

- 32 ビットの読み出し / 書き込み CP15 レジスタで、次の特徴があります。
  - – このレジスタを読み出すと、どのオーバフロー割り込みが可能かを判定できます。
  - – レジスタのビットに 1 を書き込むと、対応するオーバフロー割り込みが可能になります。
  - – レジスタのビットに 0 を書き込んでも無効です。
- 特権モードでのみアクセス可能である。

  PMINTENSET レジスタにアクセスする命令は、PMUSERENR.EN フラグが 1 にセットされている場合も含めて、ユーザモードでは常に未定義です。*c9、ユーザイネーブル レジスタ (PMUSERENR)* P. C10-117 を参照して下さい。
- セキュリティ拡張機能が実装されている場合、共通レジスタである。
- MRC または MCR コマンドで、<CRn> を c9、<opc1> を 0、<CRm> を c14、<opc2> を 1 に設定してアクセスされる。

PMINTENSET レジスタのフォーマットは次のとおりです。

31	30 … N	N – 1 … 0
C	RAZ/WI	イベントカウンタ オーバフロー割り込みイネーブルビット、P*x*、*x* は 0 ～ (N – 1)

注

PMINTENSET レジスタの説明で、N と *x* は、PMCNTENSET レジスタの説明で使用されているのと同じ意味です。*c9、カウントイネーブルセット レジスタ (PMCNTENSET)* P. C10-108 を参照して下さい。

C、ビット [31] PMCCNTR オーバフロー割り込みイネーブルビット。

このビットの読み出しおよび書き込みの動作については、P. C10-119 の表 C10-28 を参照して下さい。

ビット [30:N] RAZ/WI

P*x*、ビット [*x*]、*x* は 0 ～ (N – 1)

イベントカウンタ *x*(PMN*x*) のオーバフロー割り込みイネーブルビット。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table C10-28 shows the behavior of this bit on reads and writes.

Table C10-28 Read and write bit values for the PMINTENSET Register

Value	Meaning on read	Action on write
0	Interrupt disabled	No action, write is ignored
1	Interrupt enabled	Enable interrupt

The contents of the PMINTENSET Register are UNKNOWN on a core logic reset. To avoid spurious interrupts being generated, software must set the interrupt enable values before enabling any of the counters. Interrupts are not signaled if the PMCR.E Enable bit is set to 0.

When an interrupt is signaled, it can be removed by clearing the overflow flag for the counter in the PMOVSR Register, see *c9, Overflow Flag Status Register (PMOVSR)* on page C10-110.

Note

ARM expects that the interrupt request that can be generated on a counter overflow is also exported from the processor, meaning it can be factored into a system interrupt controller if applicable. This means that normally the system will have more levels of control of the interrupt generated.

C10.9.12 c9, Interrupt Enable Clear Register (PMINTENCLR)

The Interrupt Enable Clear Register, PMINTENCLR, disables the generation of interrupt requests on overflows from:

- the Cycle Count Register, PMCCNTR
- each implemented event counter, PMN*x*.

Reading the PMINTENCLR Register shows which overflow interrupts are enabled. Counter overflow interrupts must be enabled using the PMINTENSET Register, see *c9, Interrupt Enable Set Register (PMINTENSET)* on page C10-118.

The PMINTENCLR Register is:

- A 32-bit read/write CP15 register:
  - — reading the register shows which overflow interrupts are enabled
  - — writing a 1 to a bit of the register disables the corresponding overflow interrupt
  - — writing a 0 to a bit of the register has no effect.
- Accessible only in privileged modes.

  The instructions that access the PMINTENCLR Register are always UNDEFINED in User mode, even if the PMUSERENR.EN flag is set to 1, see *c9, User Enable Register (PMUSERENR)* on page C10-117.
- When the Security Extensions are implemented, a Common register.
- Accessed using an MRC or MCR command with <CRn> set to c9, <opc1> set to 0, <CRm> set to c14, and <opc2> set to 2.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

このビットの読み出しおよび書き込みの動作を、表 C10-28 に示します。

表 C10-28 PMINTENSET レジスタのビット値の読み出しおよび書き込み

値	読み出しの意味	書き込み時の動作
0	割り込みは不可能です。	動作は行われません。書き込みは無視されます。
1	割り込みが可能です。	割り込みを可能にします。

コアロジック リセット時の PMINTENSET レジスタの内容は未知です。偽の割り込みが生成されることを避けるため、ソフトウェアはカウンタのいずれかを可能にする前に、割り込みイネーブルの値を設定する必要があります。PMCR.E イネーブルビットが 0 にセットされている場合、割り込みは通知されません。

割り込みが通知される場合、カウンタに対応する PMOVSR レジスタのオーバフローフラグをクリアすることで削除できます。*c9、オーバフローフラグステータス レジスタ (PMOVSR)* P. C10-110 を参照して下さい。

注

カウンタのオーバフロー時に生成可能な割り込み要求は、プロセッサからエクスポートされることも予期されます。つまり、適切ならば、その割り込み要求をシステムの割り込みコントローラの要素に含めることもできます。これによって通常は、システムは生成される割り込みをより高度に制御できます。

C10.9.12 c9、割り込みイネーブルクリア レジスタ (PMINTENCLR)

割り込みイネーブルクリア レジスタ (PMINTENCLR) は、次のカウンタのオーバフローによる割り込み要求の生成を不可能にします。

- サイクルカウント レジスタ (PMCCNTR)
- 実装されている各イベントカウンタ (PMN*x*)

PMINTENCLR レジスタを読み出すと、どのオーバフロー割り込みが可能かを判定できます。カウンタオーバフロー割り込みを可能にするには、PMINTENSET レジスタを使用する必要があります。*c9、割り込みイネーブルセット レジスタ (PMINTENSET)* P. C10-118 を参照して下さい。

PMINTENCLR レジスタには、次の特徴があります。

- 32 ビットの読み出し / 書き込み CP15 レジスタで、次の特徴があります。
  - – このレジスタを読み出すと、どのオーバフロー割り込みが可能かを判定できます。
  - – レジスタのビットに 1 を書き込むと、対応するオーバフロー割り込みが不可能になります。
  - – レジスタのビットに 0 を書き込んでも無効です。
- 特権モードでのみアクセス可能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of the PMINTENCLR Register is:

31	30 … N	N-1 … 0
C	RAZ/WI	Event counter overflow interrupt disable bits, Px, for x = 0 to (N-1)

Note

In the description of the PMINTENCLR Register, N and *x* have the meanings used in the description of the PMINTENSET Register, see *c9, Interrupt Enable Set Register (PMINTENSET)* on page C10-118.

C, bit [31] PMCCNTR overflow interrupt disable bit.

See Table C10-29 for the behavior of this bit on reads and writes.

Bits [30:N] RAZ/WI.

P*m*, bit [*x*], for *x* = 0 to (N-1)

Event counter *x*, PMN*x*, overflow interrupt disable bit.

Table C10-29 shows the behavior of this bit on reads and writes.

Table C10-29 Read and write bit values for the PMINTENCLR Register

Value	Meaning on read	Action on write
0	Interrupt disabled	No action, write is ignored
1	Interrupt enabled	Disable interrupt

The contents of the PMINTENCLR Register are UNKNOWN on a core logic reset.

For more information about counter overflow interrupts see *c9, Interrupt Enable Set Register (PMINTENSET)* on page C10-118.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

PMINTENCLR レジスタにアクセスする命令は、PMUSERENR.EN フラグが 1 にセットされている場合も含めて、ユーザモードでは常に未定義です。*c9、ユーザイネーブル レジスタ (PMUSERENR)* P. C10-117 を参照して下さい。

- セキュリティ拡張機能が実装されている場合、共通レジスタです。
- MRC または MCR コマンドで、<CRn> を c9、<opc1> を 0、<CRm> を c14、<opc2> を 2 に設定してアクセスされます。

PMINTENCLR レジスタのフォーマットは次のとおりです。

31	30 … N	N − 1 … 0
C	RAZ/WI	イベントカウンタ オーバフロー割り込みディセーブル ビット、Px、x は 0 ～ (N − 1)

注

PMINTENCLR レジスタの説明で、N と *x* は、PMINTENSET レジスタの説明で使用されているのと同じ意味です。*c9、割り込みイネーブルセット レジスタ (PMINTENSET)* P. C10-118 を参照して下さい。

C、ビット [31] PMCCNTR オーバフロー割り込みディセーブルビット。

このビットの読み出しおよび書き込みの動作については、表 C10-29 を参照して下さい。

ビット [30:N] RAZ/WI

P*m*、ビット [*x*]、*x* は 0 ～ (N − 1)

イベントカウンタ *x*(PMN*x*) のオーバフロー割り込みディセーブルビット。

このビットの読み出しおよび書き込みの動作を、表 C10-29 に示します。

表 C10-29 PMINTENCLR レジスタのビット値の読み出しおよび書き込み

値	読み出しの意味	書き込み時の動作
0	割り込みは不可能です。	動作は行われません。書き込みは無視されます。
1	割り込みが可能です。	割り込みを不可能にします。

コアロジック リセット時の PMINTENCLR レジスタの内容は未知です。

カウンタオーバフロー割り込みの詳細については、*c9、割り込みイネーブルセット レジスタ (PMINTENSET)* P. C10-118 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Part D
Appendices

抄訳

パート D
付録

Appendix A
Recommended External Debug Interface

This chapter describes the recommended external debug interface. It contains the following sections:

- *System integration signals* on page AppxA-2
- *Recommended debug slave port* on page AppxA-13.

Note

This recommended external debug interface specification is not part of the ARM architecture specification. Implementers and users of the ARMv7 architecture must not consider this appendix as a requirement of the architecture. It is included as an appendix to this manual only:

- as reference material for users of ARM products that implement this interface
- as an example of how an external debug interface might be implemented.

The inclusion of this appendix is no indication of whether any ARM products might, or might not, implement this external debug interface. For details of the implemented external debug interface you must always see the appropriate product documentation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

付録 A
推奨される外部デバッグインタフェース

本章では、推奨される外部デバッグインタフェースについて説明します。本章は次のセクションから構成されています。

- *システム統合信号* P. 付録 A-2
- *推奨されるデバッグ スレーブポート* P. 付録 A-13

注

この推奨される外部デバッグインタフェース仕様は、ARM アーキテクチャ仕様の一部ではありません。ARMv7 アーキテクチャの実装者およびユーザは、この付録の内容をアーキテクチャの要件として考慮する必要はありません。本章は、次のような目的でのみ本書に付録として収録されています。

- このインタフェースを実装する ARM 製品のユーザ向けの参照資料
- 外部デバッグインタフェースが実装される方法の例

外部デバッグインタフェースがこの付録に記載されていることは、そのような外部デバッグインタフェースが ARM 製品に実装されていること、または実装されていないことを示しません。実装されている外部デバッグインタフェースの詳細については、その製品の説明書を必ず参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A.1 System integration signals

Table A-1 shows the signals recommended in v7 Debug.

Table A-1 Miscellaneous debug signals

Name	Direction	Versions	Description	Section
DBGEN	In	v6, v6.1, v7	Debug Enable	*Authentication signals* on page AppxA-3
NIDEN	In	v6, v6.1: optional v7: required	Non-Invasive Debug Enable	
SPIDEN	In	v6.1, v7	Secure Privileged Invasive Debug Enable	
SPNIDEN	In	v6.1, v7	Secure Privileged Non-Invasive Debug Enable	
DBGRESTART	In	v7 only	External restart request	*Run-control and cross-triggering signals* on page AppxA-5
DBGRESTARTED	In	v7 only	Handshake for **DBGRESTART**	
DBGTRIGGER	Out	v7 only, optional	Debug Acknowledge signal	
EDBGRQ	In	v6, v6.1, v7	External Debug Request	
DBGACK	Out	v6, v6.1, v7	Debug Acknowledge signal	*DBGACK and DBGCPUDONE* on page AppxA-7
DBGCPUDONE	Out	v7 only, optional	Debug Acknowledge signal	
COMMRX	Out	v6, v6.1, v7	DBGDTRRX full signal	*COMMRX and COMMTX* on page AppxA-9
COMMTX	Out	v6, v6.1, v7	DBGDTRTX empty signal	
DBGOSLOCKINIT	In	v7 only	Initialize OS Lock on reset	*DBGOSLOCKINIT* on page AppxA-9
DBGNOPWRDWN	Out	v6, v6.1: optional v7: required	No power-down request signal	*DBGNOPWRDWN* on page AppxA-9
DBGPWRDUP	In	v7 only	Processor powered up	*DBGPWRDUP* on page AppxA-10

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A.1 システム統合信号

v7 デバッグで推奨される信号を表 A-1 に示します。

表 A-1 その他のデバッグ信号

名前	方向	バージョン	説明	セクション
DBGEN	入力	v6、v6.1、v7	デバッグイネーブル	*認証信号* P. 付録 A-3
NIDEN	入力	v6、v6.1：オプション v7：必須	非侵襲性デバッグイネーブル	
SPIDEN	入力	v6.1、v7	セキュア特権侵襲性デバッグイネーブル	
SPNIDEN	入力	v6.1、v7	セキュア特権非侵襲性デバッグイネーブル	
DBGRESTART	入力	v7 のみ	外部からの再始動要求	*実行制御およびクロストリガ信号* P. 付録 A-5
DBGRESTARTED	入力	v7 のみ	DBGRESTART のハンドシェーク	
DBGTRIGGER	出力	v7 のみ、オプション	デバッグ応答信号	
EDBGRQ	入力	v6、v6.1、v7	外部デバッグ要求	
DBGACK	出力	v6、v6.1、v7	デバッグ応答信号	*DBGACK および DBGCPUDONE* P. 付録 A-7
DBGCPUDONE	出力	v7 のみ、オプション	デバッグ応答信号	
COMMRX	出力	v6、v6.1、v7	DBGDTRRX 満杯信号	*COMMRX および COMMTX* P. 付録 A-9
COMMTX	出力	v6、v6.1、v7	DBGDTRTX 空信号	
DBGOSLOCKINIT	入力	v7 のみ	リセット時に OS ロックを初期化する	*DBGOSLOCKINIT* P. 付録 A-9
DBGNOPWRDWN	出力	v6、v6.1：オプション v7：必須	電力オフ要求否定信号	*DBGNOPWRDWN* P. 付録 A-9
DBGPWRDUP	入力	v7 のみ	プロセッサ電力オン	*DBGPWRDUP* P. 付録 A-10

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table A-1 Miscellaneous debug signals (continued)

Name	Direction	Versions	Description	Section
DBGROMADDR[31:12]	In	v7 only	ROM Table physical address	*DBGROMADDR and DBGROMADDRV* on page AppxA-10
DBGROMADDRV	In	v7 only	ROM Table physical address valid	
DBGSELFADDR[31:12]	In	v7 only	Debug self-address offset	*DBGSELFADDR and DBGSELFADDRV* on page AppxA-10
DBGSELFADDRV	In	v7 only	Debug self-address offset valid	
DBGSWENABLE	In	v7 only	Debug software access enable	*DBGSWENABLE* on page AppxA-11
PRESETDBGn	In	v7 only	Debug logic reset	*PRESETDBGn* on page AppxA-12

A.1.1 Authentication signals

DBGEN, **NIDEN**, **SPIDEN** and **SPNIDEN** are the authentication signals.

NIDEN and **SPNIDEN** can be omitted if no non-invasive debug features are implemented.

SPIDEN and **SPNIDEN** can be omitted if Security Extensions are not implemented.

When **DBGEN** is LOW, indicating that debug is disabled:

- Halting debug events are ignored
- except for ignoring Halting debug events, the processor behaves as if DBGDSCR[15:14] == 0b00, meaning that Monitor debug-mode and Halting debug-mode are both disabled. For more information, see *Debug Status and Control Register (DBGDSCR)* on page C10-10.

For details of how these signals control enabling of invasive and non-invasive debug see Chapter C2 *Invasive Debug Authentication* and Chapter C7 *Non-invasive Debug Authentication*.

Note

The v7 Debug architecture authentication signal interface described here is compatible with the CoreSight architecture requirements for the authentication interface of a debug component. However the CoreSight architecture places additional requirements on other components in the system. For more information, see the *CoreSight Architecture Specification*.

SPIDEN also controls permissions in Debug state. For details see *Privilege in Debug state* on page C5-13.

See also *Authentication Status Register (DBGAUTHSTATUS)* on page C10-96.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 A-1 その他のデバッグ信号 （続き）

名前	方向	バージョン	説明	セクション
DBGROMADDR[31:12]	入力	v7 のみ	ROM テーブルの物理アドレス	*DBGROMADDR および DBGROMADDRV* P. 付録 A-10
DBGROMADDRV	入力	v7 のみ	ROM テーブルの物理アドレスが有効	
DBGSELFADDR[31:12]	入力	v7 のみ	デバッグの自己アドレスオフセット	*DBGSELFADDR および DBGSELFADDRV* P. 付録 A-10
DBGSELFADDRV	入力	v7 のみ	デバッグの自己アドレスオフセットが有効	
DBGSWENABLE	入力	v7 のみ	デバッグ ソフトウェアアクセスイネーブル	*DBGSWENABLE* P. 付録 A-11
PRESETDBGn	入力	v7 のみ	デバッグロジックのリセット	*PRESETDBGn* P. 付録 A-12

A.1.1 認証信号

DBGEN、NIDEN、SPIDEN、SPNIDEN は認証信号です。

NIDEN および SPNIDEN は、非侵襲性デバッグ機能が実装されていない場合は省略できます。

SPIDEN および SPNIDEN は、セキュリティ拡張機能が実装されていない場合は省略できます。

DBGEN が LOW の場合は、デバッグが不可能であることを示します。

- ホールトデバッグ イベントは無視されます。
- ホールトデバッグ イベントが無視されることを除いて、プロセッサは DBGDSCR[15:14] == 0b00 の場合と同一に動作します。これは、モニタ デバッグモードおよびホールトデバッグ モードの停止の両方が不可能であることを意味します。詳細については、*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。

これらの信号が侵襲性および非侵襲性デバッグ可能性を制御する方法の詳細については、第 C2 章 *「侵襲性デバッグの認証」*および第 C7 章 *「非侵襲性デバッグの認証」*を参照して下さい。

注

ここで説明されている v7 デバッグのアーキテクチャ認証信号インタフェースは、デバッグコンポーネントの認証インタフェースの CoreSight アーキテクチャ要件と互換性があります。ただし、CoreSight アーキテクチャでは、システムの他のコンポーネントに関する要件が追加されます。詳細については、『*CoreSight アーキテクチャ仕様*』を参照して下さい。

SPIDEN は、デバッグ状態での許可も制御します。詳細については、*デバッグ状態での特権* P. C5-13 を参照して下さい。

認証ステータスレジスタ (DBGAUTHSTATUS) P. C10-96 も参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Changing the authentication signals

In v6.1 Debug and v7 Debug, the **NIDEN**, **DBGEN**, **SPIDEN**, and **SPNIDEN** authentication signals can be controlled dynamically, meaning that they might change while the processor is running, or while the processor is in Debug state.

Note

In v6 Debug **DBGEN** is a static signal and can be changed only while the processor is in reset.

Normally, these signals are driven by the system, meaning that they are driven by a peripheral connected to the ARM processor. If the software running on the ARM processor has to change any of these signals it must follow this procedure:

1. Execute an implementation specific sequence of instructions to change the signal value. For example, this might be an instruction to write a value to a control register in a system peripheral.
2. If step 1 involves any memory operation, perform a Data Synchronization Barrier (DSB).
3. Poll the debug registers to check the signal values seen by the processor. This is required because the processor might not see the signal change until several cycles after the DSB completes.
4. Perform an Instruction Synchronization Barrier (ISB), exception entry or exception return.

The software cannot perform debug or analysis operations that rely on the new value until this procedure has been completed. The same rules apply for instructions executed through the DBGITR while in Debug state. The processor view of the authentication signals can be polled through DBGDSCR[17:16] and, in v7 Debug, the DBGAUTHSTATUS register.

Note

Exceptionally, the processor might be in Debug state even though the mode, security state and authentication signal settings are such that, in Non-debug state, debug events would be ignored. This can occur because:

- it is UNPREDICTABLE whether the behavior of debug events that are generated between a change in the authentication signals and the next Instruction Synchronization Barrier, exception entry or exception return follow the behavior of the old or new settings
- it is possible to change the authentication signals while the processor is in Debug state.

See also *Generation of debug events* on page C3-40 and *Altering CPSR privileged bits in Debug state* on page C5-14.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

認証信号の変更

v6.1 デバッグおよび v7 デバッグでは、NIDEN、DBGEN、SPIDEN、SPNIDEN 認証信号は、動的に制御できます。これは、プロセッサの実行中またはプロセッサがデバッグ状態のときに変更が可能であることを意味します。

注

v6 デバッグでは、DBGEN は静的信号で、プロセッサのリセット時にのみ変更できます。

通常、これらの信号はシステムによって駆動されます。これは、ARM プロセッサに接続されているペリフェラルによって駆動されることを意味します。ARM プロセッサで実行中のソフトウェアが、これらの信号のいずれかを変更する必要がある場合は、次の手順に従う必要があります。

1. 命令の実装固有のシーケンスを実行して、信号の値を変更します。例えば、値をシステムペリフェラルの制御レジスタに書き込む命令を実行します。
2. 手順 1 にメモリ操作が含まれている場合は、データ同期バリア (DSB) を実行します。
3. デバッグレジスタをポーリングして、プロセッサから見た信号の値をチェックします。これは、信号の変化がプロセッサから認識可能になるまで、DSB の完了後何サイクルかが必要な場合があるためです。
4. 命令同期バリア (ISB)、例外の開始、例外からの復帰のいずれかを実行します。

ソフトウェアは、この手順が完了するまで、新しい値に依存するデバッグおよび解析操作を実行することはできません。デバッグ状態で DBGITR により実行される命令にも、同じ規則が適用されます。認証信号のプロセッサからのビューは、DBGDSCR[17:16] および DBGAUTHSTATUS レジスタ（v7 デバッグの場合）を通してポーリングすることができます。

注

例外的に、モード、セキュリティ状態、認証信号が、非デバッグ状態においてデバッグイベントが無視されるように設定されている場合でも、プロセッサがデバッグ状態になることがあります。これが発生するのは次のような理由です。

- 認証信号の変化と、次の命令同期バリア、例外の開始、または例外からの復帰との間でデバッグイベントが生成された場合、その動作が古い設定と新しい設定とのどちらに従うかは予測不能である。
- プロセッサがデバッグ状態のときに、認証信号を変更することが可能である。

デバッグ イベントの生成 P. C3-40 および*デバッグ状態での CPSR 特権ビットの変更* P. C5-14 も参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A.1.2 Run-control and cross-triggering signals

ARM recommends implementation of the run-control and cross-triggering signals **EDBGRQ**, **DBGTRIGGER**, **DBGRESTART**, and **DBGRESTARTED**. These signals are particularly useful in a multiprocessor system, because using them:

- A debugger can signal a group of processors to enter Debug state.
- A debugger can signal a group of processors to leave Debug state.
- A system component can signal a group of processors to enter Debug state when any one of them enters Debug state because of a debug event on that processor. This is known as *cross-triggering*.

If you implement the recommended signalling in your system hardware, this signalling means all of the processors in the group enter or leave Debug state nearly simultaneously.

These signals can also be used in a uniprocessor implementation. For example, debug events not defined by the debug architecture might be generated externally to the processor. When one of these events occurs the external system can use these signals to cause the processor to enter Debug state. A trace macrocell might use these signals in this way.

Contact ARM for details of a recommended *Embedded Cross Trigger* (ECT) peripheral that you can use in a multiprocessor system to implement this signalling.

The following subsections describe each of the recommended signals:

- *EDBGRQ*
- *DBGTRIGGER* on page AppxA-6
- *DBGRESTART and DBGRESTARTED* on page AppxA-6.

EDBGRQ

EDBGRQ is the recommended implementation of the External Debug Request mechanism, see *Halting debug events* on page C3-38.

EDBGRQ is active-HIGH.

Once **EDBGRQ** is asserted it must be held HIGH until it is acknowledged:

- An implementation can use either **DBGACK** or **DBGTRIGGER** to acknowledge **EDBGRQ**, see:
  - *DBGACK and DBGCPUDONE* on page AppxA-7
  - *DBGTRIGGER* on page AppxA-6.
- Alternatively, debugger software might use an IMPLEMENTATION DEFINED method to acknowledge **EDBGRQ**. For example, once the processor has entered Debug state the debugger might reprogram the peripheral that is driving **EDBGRQ**.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A.1.2 実行制御およびクロストリガ信号

ARM は、実行制御およびクロストリガ信号である EDBGRQ、DBGTRIGGER、DBGRESTART、DBGRESTARTED を実装することを推奨しています。これらの信号は、次のような使用法が可能なため、特にマルチプロセッサシステムで役立ちます。

- デバッガは、プロセッサのグループに対して、デバッグ状態に遷移するよう通知できます。
- デバッガは、プロセッサのグループに対して、デバッグ状態を終了するよう通知できます。
- システムコンポーネントは、プロセッサのグループに属するプロセッサがデバッグイベントによりデバッグ状態に遷移するとき、そのグループに対してデバッグ状態に遷移するよう通知できます。これは*クロストリガ*と呼ばれます。

推奨される信号方式をシステムハードウェアに実装する場合、この信号方式はグループ内のすべてのプロセッサがほぼ同時にデバッグ状態に遷移、またはデバッグ状態を終了することを意味します。

これらの信号は、ユニプロセッサ実装でも使用できます。例えば、デバッグアーキテクチャによって定義されていないデバッグイベントを、プロセッサの外部で生成できます。これらのイベントのいずれかが発生した場合、外部システムはこれらの信号を使用してプロセッサをデバッグ状態に遷移します。トレースマクロセルでは、これらの信号をこの方法で使用できます。

マルチプロセッサシステムにこの信号方式を実装するために使用できる、推奨される*エンベデッド クロストリガ*(ECT) ペリフェラルの詳細については、ARM にお問い合わせ下さい。

次に示すサブセクションでは、推奨される信号のそれぞれについて説明します。

- *EDBGRQ*
- *DBGTRIGGER* P. 付録 A-6
- *DBGRESTART および DBGRESTARTED* P. 付録 A-6

EDBGRQ

EDBGRQ は、外部デバッグ要求機構で推奨される実装です。*ホールトデバッグ イベント* P. C3-38 を参照して下さい。

EDBGRQ はアクティブ HIGH です。

EDBGRQ がアサートされると、応答されるまで HIGH を保持する必要があります。

- 実装は、DBGACK または DBGTRIGGER のいずれかを使用して EDBGRQ に応答します。次のセクションを参照して下さい。
  - *DBGACK および DBGCPUDONE* P. 付録 A-7
  - *DBGTRIGGER* P. 付録 A-6
- また、デバッガソフトウェアは、実装定義の手法を使用して EDBGRQ に応答することもできます。例えば、プロセッサがデバッグ状態に遷移した後で、デバッガは EDBGRQ を駆動しているペリフェラルを再プログラムできます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DBGTRIGGER

The processor asserts **DBGTRIGGER** to indicate that it is committed to entering Debug state. Therefore, the system can use **DBGTRIGGER** to acknowledge **EDBGRQ**. See Chapter C5 *Debug State* for the definition of Debug state.

DBGTRIGGER is active-HIGH.

The processor must assert **DBGTRIGGER** as early as possible, so that the system can use its rising edge to signal to other devices that the processor is entering Debug state. **DBGTRIGGER** can be used for cross-triggering. For example, in a multiprocessor system, when one processor halts, the **DBGTRIGGER** signal from that processor can be used to generate an External Debug Request for the other processors.

See *DBGACK and DBGCPUDONE* on page AppxA-7 for details of the recommended External Debug Request handshaking between **EDBGRQ** and **DBGTRIGGER**.

In addition, the processor asserts **DBGTRIGGER** whenever the DBGDSCR.DBGack bit is set to 1, see *Debug Status and Control Register (DBGDSCR)* on page C10-10.

If the DBGDSCR.DBGack bit is 0, the processor deasserts **DBGTRIGGER** on exit from Debug state.

Note

Setting DBGDSCR.DBGack to 1 takes no account of the **DBGEN** and **SPIDEN** signals. Setting DBGDSCR.DBGack to 1 asserts **DBGTRIGGER** regardless of the security settings.

A v7 Debug implementation of these recommendations might not implement **DBGTRIGGER** if it would have identical behavior to **DBGACK**.

Before v7 Debug, **DBGTRIGGER** is not part of the recommended external debug interface.

DBGRESTART and DBGRESTARTED

DBGRESTART is the recommended implementation of the External Restart request, see *Leaving Debug state* on page C5-28. **DBGRESTARTED** is a handshake signal for **DBGRESTART**.

DBGRESTART and **DBGRESTARTED** are active-HIGH.

Once **DBGRESTART** is asserted, it must be held HIGH until **DBGRESTARTED** is deasserted. The processor ignores **DBGRESTART** if it is not in Debug state.

Figure A-1 on page AppxA-7 shows the four-phase handshake of **DBGRESTART** and **DBGRESTARTED**. It is diagrammatic only, and does not imply any timings.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

DBGTRIGGER

プロセッサは DBGTRIGGER をアサートして、デバッグ状態への遷移がコミットされていることを示します。このため、システムは DBGTRIGGER を使用して EDBGRQ に応答することができます。デバッグ状態の定義については、第 C5 章*「デバッグ状態」*を参照して下さい。

DBGTRIGGER はアクティブ HIGH です。

プロセッサは可能な限り早く DBGTRIGGER をアサートする必要があります。これは、システムがその立ち上がりエッジを使用して、プロセッサがデバッグ状態に遷移することを他のデバイスに通知するためです。DBGTRIGGER は、クロストリガで使用できます。例えば、マルチプロセッサシステムでは、あるプロセッサが停止した場合にそのプロセッサからの DBGTRIGGER 信号を使用して、他のプロセッサ用の外部デバッグ要求を生成することができます。

EDBGRQ と DBGTRIGGER との間の推奨される外部デバッグ要求ハンドシェークの詳細については、*DBGACK および DBGCPUDONE* P. 付録 A-7 を参照して下さい。

さらに、DBGDSCR.DBGack ビットが 1 にセットされた場合は常に、プロセッサは DBGTRIGGER をアサートします。*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。

DBGDSCR.DBGack ビットが 0 の場合、プロセッサはデバッグ状態を終了する際に DBGTRIGGER をアサート解除します。

注

DBGDSCR.DBGack を 1 にセットするとき、DBGEN および SPIDEN 信号は考慮されません。DBGDSCR.DBGack を 1 にセットすると、セキュリティ設定に関係なく DBGTRIGGER がアサートされます。

これらの推奨される v7 デバッグ実装では、DBGACK と同じ動作をする場合 DBGTRIGGER は実装されないことがあります。

v7 デバッグ以前では、DBGTRIGGER は推奨される外部デバッグインタフェースの一部ではありません。

DBGRESTART および DBGRESTARTED

DBGRESTART は、外部再始動要求の推奨される実装です。*デバッグ状態の終了* P. C5-28 を参照して下さい。DBGRESTARTED は、DBGRESTART のハンドシェーク信号です。

DBGRESTART および DBGRESTARTED はアクティブ HIGH です。

DBGRESTART がアサートされると、DBGRESTARTED がアサート解除されるまで HIGH を保持する必要があります。プロセッサは、デバッグ状態でない場合 DBGRESTART を無視します。

DBGRESTART および DBGRESTARTED のハンドシェークの 4 つのフェーズを P. 付録 A-7 の図 A-1 に示します。これは図のみで、タイミングは示されていません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure A-1 DBGRESTART / DBGRESTARTED handshake

The numbers in Figure A-1 have the following meanings:

1. If **DBGRESTARTED** is asserted HIGH the peripheral asserts **DBGRESTART** HIGH and waits for **DBGRESTARTED** to go LOW
2. The processor drives **DBGRESTARTED** LOW to deassert the signal and waits for **DBGRESTART** to go LOW
3. The peripheral drives **DBGRESTART** LOW to deassert the signal. This event indicates to the processor that it can start the transition from Debug state to Non-debug state.
4. The processor leaves Debug state and asserts **DBGRESTARTED** HIGH.

In the process of leaving Debug state the processor normally deasserts the **DBGACK**, **DBGTRIGGER**, and **DBGCPUDONE** signals. It is IMPLEMENTATION DEFINED when this change occurs relative to the changes in **DBGRESTART** and **DBGRESTARTED**.

A.1.3 DBGACK and DBGCPUDONE

DBGACK and **DBGCPUDONE** are active-HIGH.

The processor asserts **DBGACK** to indicate that it is in Debug state. Therefore, the system can use **DBGACK** as a handshake for **EDBGRQ**, instead of using **DBGTRIGGER**.

In v6 Debug and v6.1 Debug, the system can use **DBGACK** for cross-triggering.

The processor asserts **DBGCPUDONE** only after it has completed all Non-debug state memory accesses. Therefore the system can use **DBGCPUDONE** as an indicator that all memory accesses issued by the processor result from operations performed by a debugger.

Figure A-2 on page AppxA-8 shows the signalling sequence for entry to Debug state. It is diagrammatic only, and does not imply any timings.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

図 A-1 DBGRESTART/DBGRESTARTED のハンドシェーク

図 A-1 の数字には、次の意味があります。

1. DBGRESTARTED が HIGH にアサートされた場合、ペリフェラルは DBGRESTART を HIGH にアサートし、DBGRESTARTED が LOW になるのを待機します。
2. プロセッサは DBGRESTARTED を LOW に駆動して信号をアサート解除し、DBGRESTART が LOW になるのを待機します。
3. ペリフェラルは DBGRESTART を LOW に駆動して、信号をアサート解除します。このイベントは、デバッグ状態から非デバッグ状態への遷移を開始できることをプロセッサに示します。
4. プロセッサはデバッグ状態を終了し、DBGRESTARTED を HIGH にアサートします。

プロセッサは通常、デバッグ状態を終了する処理で、DBGACK、DBGTRIGGER、DBGCPUDONE 信号をアサート解除します。この変更と DBGRESTART および DBGRESTARTED の変化との相対的なタイミングは実装定義です。

A.1.3 DBGACK および DBGCPUDONE

DBGACK および DBGCPUDONE はアクティブ HIGH です。

プロセッサは DBGACK をアサートし、デバッグ状態であることを示します。このため、システムは DBGTRIGGER を使用する代わりに、DBGACK を EDBGRQ のハンドシェークとして使用することができます。

v6 デバッグおよび v6.1 デバッグでは、システムはクロストリガに DBGACK を使用することができます。

プロセッサは、非デバッグ状態でのメモリアクセスがすべて完了した後にのみ DBGCPUDONE をアサートします。このため、システムは、プロセッサによって発行されたすべてのメモリアクセスが、デバッガによって実行された操作の結果であることを示すインジケータとして、DBGCPUDONE を使用することができます。

デバッグ状態に遷移するための信号方式シーケンスを P. 付録 A-8 の図 A-2 に示します。これは図のみで、タイミングは示されていません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

See the text about the ordering of transition *a*.

Figure A-2 Signalling for Debug state entry on an External Debug Request

In Figure A-2 these events must occur in order:

1 The peripheral asserts **EBDGRQ** and waits for it to be acknowledged.

2 The processor takes the debug event and starts the Debug state entry sequence. The processor asserts **DBGTRIGGER**.

3 The processor completes the Debug state entry sequence and asserts **DBGACK**.

4 The processor completes all Non-debug state memory accesses and asserts **DBGCPUDONE**. It might do this only after intervention by an external debugger, see *Asynchronous aborts and entry to Debug state* on page C5-5.

Event *a*, the peripheral deasserting **EDBGRQ**, can occur at any time after the assertion of **EDBGRQ** is acknowledged. In the example shown in Figure A-2, the system is using **DBGTRIGGER** to acknowledge **EDBGRQ**, and therefore event *a* is not ordered relative to events 3 and 4.

In addition, the processor asserts **DBGCPUDONE** and **DBGACK** when the DBGDSCR.DBGack bit is set to 1, see *Debug Status and Control Register (DBGDSCR)* on page C10-10.

If the DBGDSCR.DBGack bit is 0, the processor deasserts **DBGCPUDONE** and **DBGACK** on exit from Debug state.

Note

Setting DBGDSCR.DBGack to 1 takes no account of the **DBGEN** and **SPIDEN** signals. Setting DBGDSCR.DBGack to 1 asserts **DBGCPUDONE** and **DBGACK** regardless of the security settings.

A v7 Debug implementation of these recommendations might not implement **DBGCPUDONE** if it would have identical behavior to **DBGACK**.

Before v7 Debug, **DBGCPUDONE** was not part of the recommended external debug interface.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

遷移状態aの順序については本文を参照して下さい。

図 A-2 外部デバッグ要求におけるデバッグ状態への遷移の信号方式

図 A-2 では、これらのイベントは次の順序で発生する必要があります。

1 ペリフェラルは EBDGRQ をアサートし、この信号への応答を待機します。

2 プロセッサはデバッグイベントを取得し、デバッグ状態への遷移シーケンスを開始します。プロセッサは DBGTRIGGER をアサートします。

3 プロセッサはデバッグ状態への遷移シーケンスを完了し、DBGACK をアサートします。

4 プロセッサはすべての非デバッグ状態のメモリアクセスを完了し、DBGCPUDONE をアサートします。これは外部デバッガによる介入後にのみ実行できます。*非同期アボートとデバッグ状態への遷移* P. C5-5 を参照して下さい。

イベント *a*（ペリフェラルによる EDBGRQ のアサート解除）は、EDBGRQ のアサートへの応答後いつでも発生することができます。図 A-2 に示されている例では、システムは DBGTRIGGER を使用して EDBGRQ に応答するため、イベント *a* とイベント 3 および 4 との相対的な順序は決定されません。

さらに、プロセッサは、DBGDSCR.DBGack ビットが 1 にセットされたときに DBGCPUDONE および DBGACK をアサートします。*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。

DBGDSCR.DBGack ビットが 0 の場合、プロセッサは、デバッグ状態を終了するときに DBGCPUDONE および DBGACK をアサート解除します。

注

DBGDSCR.DBGack を 1 にセットするとき、DBGEN および SPIDEN 信号は考慮されません。DBGDSCR.DBGack を 1 にセットすると、セキュリティ設定にかかわらず DBGCPUDONE および DBGACK がアサートされます。

これらの推奨される v7 デバッグ実装では、DBGACK と同じ動作をする場合 DBGCPUDONE が実装されないことがあります。

v7 デバッグ以前では、DBGCPUDONE は推奨される外部デバッグインタフェースの一部ではありませんでした。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A.1.4 COMMRX and COMMTX

COMMRX and **COMMTX** reflect the state of DBGDSCR[30:29] through the external debug interface:

- **COMMTX** is the inverse of DBGDSCR[29], TXfull. The processor is ready to transmit.
- **COMMRX** is equivalent to DBGDSCR[30], RXfull.

See *Debug Status and Control Register (DBGDSCR)* on page C10-10 for descriptions of the TXfull and RXfull bits.

These signals are active HIGH indicators of when the *Debug Communications Channel* (DCC) requires processing by the target system. They permit interrupt-driven communications over the DCC. By connecting these signals to an interrupt controller, software using the DCC can be interrupted whenever there is new data on the channel or when the channel is clear for transmission.

Note

There can be race conditions between reading the DCC flags through a read of DBGDSCRext and a read of the DBGDTRTXint Register or a write to the DBGDTRRXint Register through the Baseline CP14 interface. However the timing of these signals with respect to the DCC registers must be such that target code executing off an interrupt triggered by either of these signals must be able to write to DBGDTRTXint and read DBGDTRRXint without race conditions.

A.1.5 DBGOSLOCKINIT

DBGOSLOCKINIT is not required in v6 Debug and v6.1 Debug.

In v7 Debug, **DBGOSLOCKINIT** is a configuration signal that determines the state of the OS Lock immediately after a debug registers reset. On a debug registers reset:

- if **DBGOSLOCKINIT** is HIGH then the OS Lock is set from the reset
- if **DBGOSLOCKINIT** is LOW then the OS Lock is clear from the reset.

Normally, **DBGOSLOCKINIT** is tied off LOW.

For a description of debug registers reset see *Recommended reset scheme for v7 Debug* on page C6-16. For details of the OS Lock see *OS Save and Restore registers, v7 Debug only* on page C10-75.

See also *Permissions in relation to locks* on page C6-27.

A.1.6 DBGNOPWRDWN

DBGNOPWRDWN is optional in v6 Debug and v6.1 Debug.

DBGNOPWRDWN is equivalent to the value of bit [0] of the Device Power-Down and Reset Control Register. The processor power controller must work in emulate mode when this signal is HIGH.

For more information, see *Device Power-down and Reset Control Register (DBGPRCR), v7 Debug only* on page C10-31.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A.1.4 COMMRX および COMMTX

COMMRX および COMMTX には、外部デバッグインタフェースを通して DBGDSCR[30:29] の状態が反映されます。

- COMMTX は、DBGDSCR[29](TXfull) の反転です。プロセッサは送信の準備ができています。
- COMMRX は DBGDSCR[30](RXfull) と等価です。

TXfull および RXfull ビットの詳細については、*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10 を参照して下さい。

これらの信号は、*デバッグ通信チャネル* (DCC) でターゲットシステムによる処理が必要になる場合のアクティブ HIGH インジケータです。DCC を経由する割り込み駆動型通信が可能です。これらの信号を割り込みコントローラに接続することにより、チャネル上に新規データが存在する、またはチャネルがクリアで転送が可能な場合であればいつでも、DCC を使用するソフトウェアに割り込みをかけることができます。

注

DBGDSCRext の読み出しによる DCC フラグ読み出しと、ベースライン CP14 インタフェースを経由する DBGDTRTXint レジスタの読み出しまたは DBGDTRRXint レジスタへの書き込みとの間に、競合状態が存在する場合があります。ただし、DCC レジスタに対するこれらの信号のタイミングでは、これらの信号のいずれかによってトリガされる割り込みによって実行されるターゲットコードが、競合状態なしで DBGDTRTXint への書き込みおよび DBGDTRRXint の読み出しを実行可能な必要があります。

A.1.5 DBGOSLOCKINIT

DBGOSLOCKINIT は、v6 デバッグおよび v6.1 デバッグでは必要ありません。

v7 デバッグでは、DBGOSLOCKINIT は、デバッグレジスタのリセット直後の OS ロックの状態を決定する構成信号です。この信号は、デバッグレジスタのリセット時に次のように設定されます。

- DBGOSLOCKINIT が HIGH の場合、リセット時に OS ロックが設定されます。
- DBGOSLOCKINIT が LOW の場合、リセット時に OS ロックがクリアされます。

通常、DBGOSLOCKINIT は LOW に固定されます。

デバッグレジスタのリセットの詳細については、*v7 デバッグで推奨されるリセット方式* P. C6-16 を参照して下さい。OS ロックの詳細については、*OS 保存および復元レジスタ（v7 デバッグのみ）* P. C10-75 を参照して下さい。

ロックとアクセス許可との関係 P. C6-26 も参照して下さい。

A.1.6 DBGNOPWRDWN

DBGNOPWRDWN は、v6 デバッグおよび v6.1 デバッグではオプションです。

DBGNOPWRDWN は、デバイス電力オフおよびリセット制御レジスタのビット [0] の値と等価です。プロセッサの電力コントローラは、この信号が HIGH のときはエミュレートモードで動作する必要があります。

詳細については、*デバイス電力オフおよびリセット制御レジスタ (DBGPRCR)、v7 デバッグのみ* P. C10-31 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A.1.7 DBGPWRDUP

DBGPWRDUP is not required in v6 Debug and v6.1 Debug.

DBGPWRDUP is not required in a SinglePower system, that is, it is not required in a design that has only one power domain.

The **DBGPWRDUP** input signal is HIGH when the processor is powered up, and LOW otherwise. The **DBGPWRDUP** signal is reflected in bit [0] of the Device Power-Down and Reset Status Register.

See also *Device Power-down and Reset Status Register (DBGPRSR), v7 Debug only* on page C10-34 and *Permissions in relation to power-down* on page C6-28.

A.1.8 DBGROMADDR and DBGROMADDRV

DBGROMADDR and **DBGROMADDRV** are not required in v6 Debug and v6.1 Debug. They are required in v7 Debug if the memory-mapped interface is implemented.

DBGROMADDR specifies bits [31:12] of the ROM Table table physical address. This is a configuration input. It must be either:

- be a tie-off
- change only while the processor is in reset.

In a system with multiple ROM Tables, this address must be tied off to the top-level ROM Table address.

In a system with no ROM Table this address must be tied off with the physical address where the debug registers are memory-mapped. Debug software can use the debug component identification registers at the end of the 4KB block addressed by **DBGROMADDR** to distinguish a ROM table from a processor.

Note

If the system implements more than one debug component, for example a processor and a trace macrocell, a ROM Table must be provided.

DBGROMADDRV is the valid signal for **DBGROMADDR**. If the address cannot be determined, **DBGROMADDR** must be tied off to zero and **DBGROMADDRV** tied LOW.

The format of ROM Tables is defined in the *ARM Debug Interface v5 Architecture Specification*.

A.1.9 DBGSELFADDR and DBGSELFADDRV

DBGSELFADDR and **DBGSELFADDRV** are not required in v6 Debug and v6.1 Debug.

In v7 Debug, **DBGSELFADDR** and **DBGSELFDDRV** are required if the memory-mapped interface is implemented. If **DBGROMADDR** and **DBGROMADDRV** are not implemented, **DBGSELFADDR** and **DBGSELFADDRV** must not be implemented.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A.1.7 DBGPWRDUP

DBGPWRDUP は、v6 デバッグおよび v6.1 デバッグでは必要ありません。

DBGPWRDUP は、SinglePower システム、つまり 1 つの電力ドメインのみを持つ設計では必要ありません。

DBGPWRDUP 入力信号は、プロセッサの電力がオンの場合は HIGH で、そうでない場合は LOW です。DBGPWRDUP 信号は、デバイス電力オフおよびリセットステータス レジスタのビット [0] に反映されます。

デバイス電力オフおよびリセットステータス レジスタ (DBGPRSR)、v7 デバッグのみ P. C10-34 および*電力オフとアクセス許可との関係* P. C6-27 も参照して下さい。

A.1.8 DBGROMADDR および DBGROMADDRV

DBGROMADDR および DBGROMADDRV は、v6 デバッグおよび v6.1 デバッグでは必要ありません。これらは、v7 デバッグでメモリマップされたインタフェースが実装されている場合に必要となります。

DBGROMADDR は、ROM テーブルのテーブル物理アドレスのビット [31:12] を指定します。これは構成入力です。次のいずれかの状態にする必要があります。

- 固定されている。
- プロセッサのリセット時にのみ変化する。

複数の ROM テーブルを持つシステムでは、このアドレスはトップレベルの ROM テーブルアドレスに固定する必要があります。

ROM テーブルを持たないシステムでは、このアドレスはデバッグレジスタがメモリマップされている物理アドレスに固定する必要があります。デバッグソフトウェアでは、DBGROMADDR によって指定された 4KB ブロックの最後にあるデバッグコンポーネント識別レジスタを使用して、ROM テーブルをプロセッサから区別することができます。

注

システムに、プロセッサとトレースマクロセルなど、複数のデバッグコンポーネントを実装する場合、ROM テーブルを搭載する必要があります。

DBGROMADDRV は、DBGROMADDR が有効であることを示す信号です。アドレスを決定することができない場合は、DBGROMADDR は 0 に固定し、DBGROMADDRV は LOW に固定する必要があります。

ROM テーブルのフォーマットは、『*ARM デバッグインタフェース v5、アーキテクチャ仕様書*』で定義されています。

A.1.9 DBGSELFADDR および DBGSELFADDRV

DBGSELFADDR および DBGSELFADDRV は、v6 デバッグおよび v6.1 デバッグでは必要ありません。

v7 デバッグでは、DBGSELFADDR および DBGSELFADDRV は、メモリマップされたインタフェースが実装されている場合に必要になります。DBGROMADDR および DBGROMADDRV が実装されていない場合、DBGSELFADDR および DBGSELFADDRV を実装できません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DBGSELFADDR specifies bits [31:12] of the two's complement signed offset from the ROM Table physical address to the physical address where the debug registers are Memory-mapped. This is a configuration input. It must either:

- be a tie-off
- change only while the processor is in reset.

If there is no ROM Table, **DBGROMADDR** must be configured as described in the section *DBGROMADDR and DBGROMADDRV* on page AppxA-10, and **DBGSELFADDR** must be tied off to zero with **DBGSELFADDRV** tied HIGH.

DBGSELFADDRV is the valid signal for **DBGSELFADDR**. If the offset cannot be determined, **DBGSELFADDR** must be tied off to zero and **DBGSELFADDRV** tied LOW.

A.1.10 DBGSWENABLE

DBGSWENABLE is not required in v6 Debug and v6.1 Debug.

In v7 Debug, **DBGSWENABLE** is driven by the Debug Access Port. For details see the *ARM Debug Interface v5 Architecture Specification*.

DBGSWENABLE is an active-HIGH signal that must be asserted to enable system access to the debug register file. That is, if deasserted it prevents access through the memory-mapped and Extended CP14 interfaces. This gives the debugger full control over the debug registers in the processor.

When this signal is deasserted by the debugger by a means that is IMPLEMENTATION DEFINED, memory-mapped interface accesses return an error response and most Extended CP14 operations become UNDEFINED instructions. See *CP14 debug registers access permissions* on page C6-36 and *Permission summaries for memory-mapped and external debug interfaces* on page C6-45.

In the ARM Debug Interface v5, **DBGSWENABLE** is asserted by setting the DbgSwEnable control bit in the access port *Control Status Word Register* (CSW) to 1. For the memory-mapped interface, when the DbgSwEnable control bit is set to 0 the generation of slave-generated errors is a function of the ADIv5 Debug Access Port, and therefore the processor ignores the **DBGSWENABLE** signal for the memory-mapped interface. For details see the *ARM Debug Interface v5 Architecture Specification*.

The **DBGSWENABLE** signal has no effect on accesses through the external debug interface.

Normally, the **DBGSWENABLE** signal must be asserted at debug logic reset and deasserted under debugger control.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DBGSELFADDR には、ROM テーブルの物理アドレスから、デバッグレジスタがメモリマップされている物理アドレスへの、2 の補数形式による符号付きオフセットのビット [31:12] を指定します。これは構成入力です。次のいずれかの状態にする必要があります。

- 固定されている。
- プロセッサのリセット時にのみ変化する。

ROM テーブルが存在しない場合、DBGROMADDR はセクション *DBGROMADDR および DBGROMADDRV* P. 付録 A- 10 の説明に従って構成する必要があり、DBGSELFADDR は 0 に固定し、DBGSELFADDRV は HIGH に固定する必要があります。

DBGSELFADDRV は、DBGSELFADDR が有効であることを示す信号です。オフセットを決定できない場合、DBGSELFADDR は 0 に固定し、DBGSELFADDRV は LOW に固定する必要があります。

A.1.10 DBGSWENABLE

DBGSWENABLE は、v6 デバッグおよび v6.1 デバッグでは必要ありません。

v7 デバッグでは、DBGSWENABLE はデバッグアクセス ポートによって駆動されます。詳細については、『*ARM デバッグインタフェース v5、アーキテクチャ仕様書*』を参照して下さい。

DBGSWENABLE はアクティブ HIGH 信号で、システムがデバッグレジスタ ファイルにアクセスすることを許可するためにアサートする必要があります。つまり、アサート解除されている場合、メモリマップされたインタフェースおよび拡張 CP14 インタフェースを通してアクセスすることはできません。これにより、デバッガはプロセッサのデバッグレジスタを完全に制御できます。

デバッガにより、この信号が実装定義の方法によってアサート解除された場合は、メモリマップされたインタフェースへのアクセスでエラー応答が返され、大部分の拡張 CP14 操作が未定義命令となります。*CP14 デバッグレジスタのアクセス許可* P. C6- 36 および*メモリマップおよび外部デバッグインタフェースのアクセス許可の概要* P. C6- 45 を参照して下さい。

ARM デバッグインタフェース v5 では、DBGSWENABLE は、アクセスポート*制御ステータスワード レジスタ*(CSW)の DbgSwEnable 制御ビットを 1 にセットすることによりアサートされます。メモリマップされたインタフェースの場合は、DbgSwEnable 制御ビットが 0 にセットされているときにはスレーブ生成エラーの生成が ADIv5 デバッグアクセス ポートの機能になるため、プロセッサはメモリマップされたインタフェースの DBGSWENABLE 信号を無視します。詳細については、『*ARM デバッグインタフェース v5、アーキテクチャ仕様書*』を参照して下さい。

DBGSWENABLE 信号は、外部デバッグインタフェースを通したアクセスには影響を与えません。

通常、DBGSWENABLE 信号は、デバッグロジックのリセット時にアサートし、デバッガ制御の下でアサート解除する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A.1.11 PRESETDBGn

PRESETDBGn is not required in v6 Debug and v6.1 Debug. The debug logic is only reset on system power-up reset.

The reset signal resets all debug registers. See also *Recommended reset scheme for v7 Debug* on page C6-16.

Note

Do not use the **PRESETDBGn** signal to reset the debug registers if the debug system is connected to a debug monitor that uses the CP14 debug interface.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

A.1.11 PRESETDBGn

PRESETDBGn は、v6 デバッグおよび v6.1 デバッグでは必要ありません。デバッグロジックは、システムの電力オンリセット時にのみリセットされます。

リセット信号により、すべてのデバッグレジスタがリセットされます。*v7 デバッグで推奨されるリセット方式* P. C6- 16 も参照して下さい。

注

デバッグシステムが CP14 デバッグインタフェースを使用するデバッグモニタに接続されている場合は、PRESETDBGn 信号を使用してデバッグレジスタをリセットしないで下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A.2 Recommended debug slave port

This slave port is not required in v6 Debug and v6.1 Debug.

The memory-mapped interface is optional on v7 Debug. This section describes the recommended APBv3 slave port. It provides both the memory-mapped and external debug interfaces.

A valid external debug interface for v7 Debug is any access mechanism that enables the external debugger to complete reads or writes to the memory-mapped registers described in *The memory-mapped and recommended external debug interfaces* on page C6-43.

In v7 Debug a memory-mapped interface can be implemented to provide access to the debug registers using load and store operations. Such an interface is sufficient for the requirements of the external debug interface, and therefore it is possible to implement both the memory-mapped and external debug interfaces using a single memory slave port on the processor.

This section describes the v7 Debug recommendations for an APBv3 memory slave port APBv3 as part of the external debug interface. In addition, ARM recommends a Debug Access Port capable of mastering an APBv3 bus and compatible with the *ARM Debug Interface v5* (ADIv5). Figure A-3 shows the recommendations.

Figure A-3 Recommended external debug interface, including APBv3 slave port

In Figure A-3, signals with a lower-case n suffix are active LOW and all other signals are active HIGH.

ARM recommends that the debug registers are accessible through an ARM AMBA 3 Peripheral Bus version 1 (APBv3) external debug interface. This APBv3 interface:

- is 32 bits wide
- supports only 32-bit reads and writes

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A.2 推奨されるデバッグ スレーブポート

このスレーブポートは、v6 デバッグおよび v6.1 デバッグでは必要ありません。

メモリマップされたインタフェースは、v7 デバッグではオプションです。このセクションでは、推奨される APBv3 スレーブポートについて説明します。このスレーブポートでは、メモリマップされたインタフェースと外部デバッグインタフェースの両方が提供されます。

v7 デバッグで有効な外部デバッグインタフェースは、外部デバッガが、*メモリマップされたインタフェースと推奨される外部デバッグインタフェース* P. C6-43 で説明されているメモリマップされたレジスタとの読み出しまたは書き込みを完了することができる、任意のアクセス機構です。

v7 デバッグでは、メモリマップされたインタフェースを実装し、ロードおよびストア操作を使用してデバッグレジスタにアクセスすることができます。このようなインタフェースは外部デバッグインタフェースの要件を十分に満たしているため、プロセッサ上の単一のメモリ スレーブポートを使用して、メモリマップされたインタフェースおよび外部デバッグインタフェースの両方を実装することができます。

このセクションでは、外部デバッグインタフェースの一部として、APBv3 メモリ スレーブポート APBv3 の v7 デバッグ推奨事項について説明します。さらに、APBv3 バスのマスタとして動作でき、*ARM デバッグインタフェース v5*(ADIv5) と互換性のあるデバッグ アクセスポートが推奨されます。推奨事項を図 A-3 に示します。

図 A-3 推奨される外部デバッグインタフェース（APBv3 スレーブポートを含む）

図 A-3 では、小文字の接尾文字 n で終わる信号はアクティブ LOW で、他のすべての信号はアクティブ HIGH です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- has stallable accesses
- has slave-generated aborts
- has 10 address bits ([11:2]) mapping 4KB of memory.

An extra signal, **PADDRDBG[31]**, informs the debug slave port of the source of the access, as shown in Table A-2.

Table A-2 lists the external debug interface signals.

Table A-2 Recommended external debug interface signals

Name	Direction	Description
PSELDBG	In	Selects the external debug interface
PADDRDBG[31,11:2]	In	Address. see *PADDRDBG* on page AppxA-15
PRDATADBG[31:0]	Out	Read data
PWDATADBG[31:0]	In	Write data
PENABLEDBG	In	Indicates a second and subsequent cycle of a transfer
PREADYDBG	Out	Used to extend a transfer, by inserting wait states
PSLVERRDBG	Out	Slave-generated error response, see *PSLVERRDBG* on page AppxA-15
PWRITEDBG	In	Distinguishes between a read (LOW) and a write (HIGH)
PCLKDBG	In	Clock
PCLKENDBG	In	Clock enable for **PCLKDBG**

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

デバッグレジスタは、ARM AMBA 3 ペリフェラルバス バージョン 1(APBv3) 外部デバッグインタフェース経由でアクセス可能にすることをお勧めします。この APBv3 インタフェースには、次の特徴があります。

- 32 ビット幅
- 32 ビットの読み出しおよび書き込みのみをサポート
- アクセスが停止可能
- スレーブ生成アボート
- 4KB のメモリをマッピングする 10 個のアドレスビット ([11:2])

表 A-2 に示すように、外部信号 PADDRDBG[31] は、デバッグ スレーブポートにアクセスのソースを通知します。

外部デバッグインタフェース信号の一覧を、表 A-2 に示します。

表 A-2 推奨される外部デバッグインタフェース信号

名前	方向	説明
PSELDBG	入力	外部デバッグインタフェースを選択
PADDRDBG[31、11:2]	入力	アドレス（*PADDRDBG* P. 付録 A-15 を参照）
PRDATADBG[31:0]	出力	読み出しデータ
PWDATADBG[31:0]	入力	データの書き込み
PENABLEDBG	入力	転送の 2 番目およびそれ以後のサイクルであることを示す
PREADYDBG	出力	ウェイトステートを挿入して、転送を拡張するために使用
PSLVERRDBG	出力	スレーブ生成エラー応答（*PSLVERRDBG* P. 付録 A-15 を参照）
PWRITEDBG	入力	読み出し (LOW) と書き込み (HIGH) とを区別する
PCLKDBG	入力	クロック
PCLKENDBG	入力	PCLKDBG のクロックイネーブル

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A.2.1 PADDRDBG

PADDRDBG selects the register to read or write.

In the recommended debug slave port that implements both the external debug interface and the memory-mapped interface, the complete register set is aliased twice:

- the first view, the memory-mapped interface view, starts at `0x0`
- the second view, the external debug interface view, starts at `0x80000000`.

This means that **PADDRDBG**[31] is used to distinguish the source of an access:

PADDRDBG[31] == 0 Access from system

PADDRDBG[31] == 1 Access from external debugger.

Note

The only bits of **PADDRDBG** that are specified are **PADDRDBG[31, 11:2]**. Bits [1:0] are not required because all registers are word-sized, and bit [31] is used as described to indicate the source of the access. Because some HDL languages do not permit partial buses to be specified in this way an actual implementation might use a different name for **PADDRDBG[31]**, such as **PADDRDBG31**.

A.2.2 PSLVERRDBG

PSLVERRDBG is used to signal an aborted access.

PSLVERRDBG has the same timing as the ready response, **PREADYDBG**. Under the v7 Debug model, accesses are only aborted, by asserting **PSLVERRDBG** HIGH, in a situation related to power-down. See also *Permission summaries for memory-mapped and external debug interfaces* on page C6-45.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A.2.1 PADDRDBG

PADDRDBG は、レジスタの読み出しまたは書き込みを選択します。

外部デバッグインタフェースおよびメモリマップされたインタフェースの両方を実装する、推奨のデバッグ スレーブポートでは、完全なレジスタセットのエイリアスが 2 回作成されます。

- 最初のビューはメモリマップされたインタフェースのビューで、`0x0` から始まります。
- 2 番目のビューは外部デバッグインタフェースのビューで、`0x80000000` から始まります。

これは、PADDRDBG[31] が、アクセスのソースの区別に使用されることを意味します。

PADDRDBG[31] == 0 システムからのアクセス

PADDRDBG[31] == 1 外部デバッガからのアクセス

注

PADDRDBG のうち、指定されるビットは PADDRDBG[31、11:2] のみです。すべてのレジスタがワードサイズであるため、ビット [1:0] は必要ありません。ビット [31] は、アクセスのソースを示すため、説明されているように使われます。一部の HDL 言語では、この方法で部分的なバスを指定することが許可されていないため、実際の実装では PADDRDBG[31] に対して PADDRDBG31 などの別の名前を使用することがあります。

A.2.2 PSLVERRDBG

PSLVERRDBG は、アボートされたアクセスを通知するために使用されます。

PSLVERRDBG は、準備完了応答 PREADYDBG と同じタイミングになります。v7 デバッグモデルの電力オフに関連する状況では、PSLVERRDBG を HIGH にアサートすることによってのみ、アクセスがアボートされます。*メモリマップおよび外部デバッグインタフェースのアクセス許可の概要* P. C6- 45 も参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Appendix B
Common VFP Subarchitecture Specification

This appendix describes version 2 of the Common VFP subarchitecture. It contains the following sections:

- *Scope of this appendix* on page AppxB-2
- *Introduction to the Common VFP subarchitecture* on page AppxB-3
- *Exception processing* on page AppxB-6
- *Support code requirements* on page AppxB-11
- *Context switching* on page AppxB-14
- *Subarchitecture additions to the VFP system registers* on page AppxB-15
- *Version 1 of the Common VFP subarchitecture* on page AppxB-23.

Note

This VFP subarchitecture specification is not part of the ARM architecture specification. Implementers and users of the ARMv7 architecture must not consider this appendix as a requirement of the architecture. It is included as an appendix to this manual only:

- as reference material for users of ARM VFP products that implement this subarchitecture
- as an example of how a VFP subarchitecture might be implemented.

The inclusion of this appendix is no indication of whether any ARMv7 VFP implementations by ARM might, or might not, implement this Common VFP subarchitecture. For details of the implemented VFP subarchitecture you must always see the appropriate product documentation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

付録 B
一般的な VFP サブアーキテクチャ仕様

この付録では、一般的な VFP アーキテクチャのバージョン 2 について説明します。本章は次のセクションから構成されています。

- *この付録の範囲* P. 付録 B-2
- *一般的な VFP サブアーキテクチャの概要* P. 付録 B-3
- *例外処理* P. 付録 B-6
- *サポートコードの要件* P. 付録 B-11
- *コンテキストスイッチ* P. 付録 B-14
- *VFP システムレジスタへのサブアーキテクチャ固有の追加* P. 付録 B-15
- *一般的な VFP サブアーキテクチャのバージョン 1* P. 付録 B-23

注

この VFP サブアーキテクチャ仕様は、ARM アーキテクチャ仕様の一部ではありません。ARMv7 アーキテクチャの実装者およびユーザは、この付録の内容をアーキテクチャの要件として考慮する必要はありません。本章は、次のような目的でのみ本書に付録として収録されています。

- このサブアーキテクチャを実装する ARM VFP 製品のユーザに対する参照資料。
- VFP サブアーキテクチャの実装方法の例。

一般的な VFP サブアーキテクチャがこの付録に記載されていることは、ARM 製の ARMv7 VFP 実装にそのような一般的な VFP サブアーキテクチャが実装されていること、または実装されていないことを示しません。実装されている VFP サブアーキテクチャの詳細については、その製品の説明書を必ず参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.1 Scope of this appendix

This specification describes the Common VFP subarchitecture. This is not part of the ARMv7 architecture specification, see the *Note* on the cover page of this appendix.

The Common VFP subarchitecture is an interface provided by VFP coprocessor hardware to support code in an operating system.

This appendix is for engineers implementing and validating a VFP coprocessor, and for engineers implementing support code in an operating system.

The main sections of this appendix describe version 3 of the Common VFP subarchitecture. Version 3 is an extension to the previously-published version 2 of the subarchitecture. Version 3 of the Common VFP subarchitecture includes more support for synchronous exception reporting.

Support code for version 1 of the subarchitecture differs from version 2 only when trapped exception handling of the Inexact exception is enabled. The differences from version 1 of the subarchitecture are described in *Version 1 of the Common VFP subarchitecture* on page AppxB-23.

The differences between versions 2 and 3 of the subarchitecture are described in *Version 2 of the Common VFP subarchitecture* on page AppxB-24.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B.1 この付録の範囲

この仕様では、一般的な VFP サブアーキテクチャについて説明します。これは、ARMv7 アーキテクチャ仕様の一部ではありません。この付録の表紙の*注*を参照して下さい。

この一般的な VFP サブアーキテクチャは、VFP コプロセッサハードウェアによってオペレーティングシステムのサポートコードに提供されるインタフェースです。

この付録は、VFP コプロセッサの実装および検証を行うエンジニア、およびオペレーティングシステムにサポートコードを実装するエンジニア向けに記述されたものです。

この付録の主なセクションでは、一般的な VFP サブアーキテクチャのバージョン 3 について説明します。バージョン 3 は、サブアーキテクチャの以前に公開されたバージョン 2 を拡張したものです。一般的な VFP サブアーキテクチャのバージョン 3 では、同期例外レポートのサポートが拡張されています。

サブアーキテクチャのバージョン 1 用のサポートコードは、不正確な例外のトラップ例外処理が可能な場合のみバージョン 2 と異なります。サブアーキテクチャのバージョン 1 との違いは、*一般的な VFP サブアーキテクチャのバージョン 1* P. 付録 B-23 で説明されています。

サブアーキテクチャのバージョン 2 と 3 との違いは、*一般的な VFP サブアーキテクチャのバージョン 2* P. 付録 B-24 で説明されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.2 Introduction to the Common VFP subarchitecture

The VFP architecture describes the interface provided by a VFP implementation to application software. A complete implementation of the VFP architecture might include both a hardware coprocessor and a software component, known as the support code. Support code must signal trapped floating-point exceptions to application software, and provide other implementation-dependent functions. The Common VFP subarchitecture describes an interface between VFP coprocessor hardware and support code.

B.2.1 VFP support code and bounced instructions

Support code is entered through the ARM Undefined Instruction exception vector, when the VFP hardware does not respond to a VFP instruction. This software entry is known as a *bounce*.

The bounce mechanism supports trapped floating-point exceptions. Trapped floating-point exceptions, known as *traps*, are floating-point exceptions that an implementation must pass back for application software to resolve. See *Trapped floating-point exception handling* on page AppxB-10.

Support code might perform other tasks, in addition to trap handler calls. These tasks are determined by the implementation. Typically, additional support code functions might handle rare conditions that are either difficult to implement in hardware, or gate-intensive in hardware. This approach permits software behavior to be consistent across implementations with varying degrees of hardware support.

B.2.2 Exception processing terminology

A condition that causes a VFP instruction to call support code is called an *exceptional condition*.

The VFP instruction that contains the floating-point operation requiring support code is known as the exception-generating instruction.

The VFP instruction that causes a bounce to occur is known as the trigger instruction.

An implementation can use both synchronous and asynchronous exception signaling:

- if an exception is signaled synchronously, the exception-generating instruction is also the trigger instruction.
- if an exception is signaled asynchronously, the trigger instruction is a VFP instruction that occurs after the exception-generating instruction.

An implementation can issue and complete additional VFP instructions before bouncing the trigger instruction.

An implementation can issue a maximum of one additional VFP instruction that it cannot complete. This instruction is known as the bypassed instruction. This instruction is retired in the ARM processor and cannot be reissued. Therefore, it must be executed by the VFP support code.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.2 一般的な VFP サブアーキテクチャの概要

VFP アーキテクチャでは、VFP 実装によってアプリケーションソフトウェアに提供されるインタフェースが記述されます。VFP アーキテクチャの完全な実装には、ハードウェアコプロセッサと、サポートコードと呼ばれるソフトウェアコンポーネントの両方が含まれます。サポートコードは、トラップされた浮動小数点例外をアプリケーションソフトウェアに信号で通知する必要があり、他の実装依存の機能を提供します。一般的な VFP サブアーキテクチャでは、VFP コプロセッサハードウェアとサポートコードとの間のインタフェースが記述されます。

B.2.1 VFP サポートコードとバウンスされた命令

サポートコードは、VFP ハードウェアが VFP 命令に応答しない場合に、ARM 未定義命令例外ベクタから開始されます。この方法でソフトウェアを開始することを、*バウンス*と呼びます。

バウンス機構では、トラップされた浮動小数点例外がサポートされます。トラップされた浮動小数点例外は*トラップ*と呼ばれますが、これは実装がアプリケーションソフトウェアに戻して解決する必要がある浮動小数点例外です。*トラップされた浮動小数点例外処理* P. 付録 B-10 を参照して下さい。

サポートコードは、トラップハンドラ呼び出しに加えて、他のタスクも実行することがあります。これらのタスクは、実装によって決定されます。一般に、追加のサポートコード機能では、ハードウェアに実装することが困難である、またはハードウェアで実装すると大量のゲートを使用する、稀な状況の処理が行われます。この手法により、ハードウェアによるサポートの度合いが違っても、ソフトウェアの動作が実装間で整合することができます。

B.2.2 例外処理の用語

VFP 命令がサポートコードを呼び出す条件は、*例外条件*と呼ばれます。

サポートコードを必要とする浮動小数点操作が含まれる VFP 命令は、例外生成命令と呼ばれます。

バウンスを発生させる VFP 命令は、トリガ命令と呼ばれます。

実装では、次のようにして同期および非同期例外の両方の通知を使用できます。

- 例外が同期して通知される場合、例外生成命令もトリガ命令です。
- 例外が非同期で通知される場合、トリガ命令は例外生成命令後に発生する VFP 命令です。

実装では、トリガ命令をバウンスする前に、追加の VFP 命令の発行および完了を行うことができます。

実装は、完了することができない最大 1 つの追加 VFP 命令を発行することができます。この命令は、バイパスされた命令と呼ばれます。この命令は ARM プロセッサ内で終了しており、再発行することはできません。このため、この命令は VFP サポートコードによって実行する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.2.3 Hardware and software implementation

The Common VFP subarchitecture requires the VFP hardware implementation to perform completely all load, store and register transfer instructions. These instructions cannot generate floating-point exceptions.

The division of labor between the hardware and software components of a VFP implementation for CDP operations is IMPLEMENTATION DEFINED.

Typically, the hardware handles all common cases, to optimize performance. When the hardware encounters a case that it cannot handle on its own it calls the software component, the support code for the hardware, to deal with it.

For more information, see *Advanced SIMD and VFP extensions* on page A2-20.

B.2.4 VFP subarchitecture system registers

The Common VFP subarchitecture adds two instruction registers:

- for asynchronous exceptions, the FPINST register contains the exception-generating instruction
- the FPINST2 register contains the bypassed instruction, if there is one.

Both instruction registers are optional:

- The FPINST register is required only if at least one supported configuration can bounce instructions asynchronously.
- The FPINST2 register is required only if the processor can commit to issuing a VFP instruction before an exceptional case is detected in an earlier VFP instruction.

The Common VFP subarchitecture adds new fields to the FPEXC Register:

- the FPEXC.VECITR field contains an encoding that gives the remaining vector length of the exception-generating instruction
- the FPEXC.FP2V bit indicates if the FPINST2 register contains an instruction that the support code must execute
- the FPEXC.DEX bit is set when a synchronous bounce is caused by a floating-point exception, indicating that the support code must execute the bounced instruction
- the FPEXC.VV bit is set when a synchronous bounce is caused by a floating-point exception, and the FPEXC.VECITR field is valid
- an IMPLEMENTATION DEFINED field, for the implementation to give more information about the exceptional condition that caused the bounce.

See *The Floating-Point Exception Register (FPEXC)* on page B1-68 for a description of the minimum implementation of the FPEXC required by the VFP architecture.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B.2.3 ハードウェアおよびソフトウェア実装

一般的な VFP サブアーキテクチャでは、すべてのロード、ストア、およびレジスタ転送命令を完全に実行するため、VFP ハードウェア実装が必要です。これらの命令は、浮動小数点例外を生成することはできません。

CDP 操作用の VFP 実装のハードウェアおよびソフトウェアコンポーネントの間の処理の分担は、実装定義です。

通常、ハードウェアはすべての一般的なケースを処理し、パフォーマンスを最適化します。ハードウェアだけで処理することができない事態が発生した場合、ハードウェアは、ハードウェア用のサポートコードであるソフトウェアコンポーネントを呼び出し、その事態を処理します。

詳細については、*アドバンスト SIMD および VFP 拡張機能* P. A2-20 を参照して下さい。

B.2.4 VFP サブアーキテクチャのシステムレジスタ

一般的な VFP サブアーキテクチャでは、次の 2 つの命令レジスタが追加されます。

- 非同期例外の場合、FPINST レジスタに例外生成命令が格納されます。
- FPINST2 レジスタには、バイパスされた命令が存在する場合に、その命令が格納されます。

次に示す命令レジスタは、いずれもオプションです。

- 少なくとも 1 つのサポートされる構成が、非同期で命令をバウンスすることができる場合にのみ、FPINST レジスタが必要です。
- 以前の VFP 命令で例外的な事態が検出される前に、プロセッサが VFP 命令の発行をコミットすることができる場合にのみ、FPINST2 レジスタが必要です。

一般的な VFP サブアーキテクチャでは、FPEXC レジスタに次の新規フィールドが追加されます。

- FPEXC.VECITR フィールドには、例外生成命令の残りのベクタ長を指定するエンコードが格納されます。
- FPEXC.FP2V ビットは、サポートコードが実行する必要がある命令が FPINST2 レジスタに格納されているかどうかを示します。
- FPEXC.DEX ビットは、浮動小数点例外により同期バウンスが引き起こされた場合にセットされます。これは、サポートコードがバウンスされた命令を実行する必要があることを示します。
- FPEXC.VV ビットは、浮動小数点例外により同期バウンスが引き起こされた場合にセットされます。この場合、FPEXC.VECITR フィールドは有効です。
- 実装定義フィールドは、バウンスを引き起こした例外的な条件に関する詳細情報を実装で提供するために使用されます。

VFP アーキテクチャで必要になる FPEXC の最小限の実装に関する説明については、*浮動小数点例外レジスタ (FPEXC)* P. B1-68 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

In version 2 of the Common VFP subarchitecture the FPEXC.EX bit is set to 1 only when an asynchronous bounce occurs.

Software can detect the presence of the instruction registers by testing the FPEXC.EX and FPEXC.FP2V bits, as described in *Detecting which VFP Common subarchitecture registers are implemented* on page AppxB-22.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

一般的な VFP サブアーキテクチャのバージョン 2 では、FPEXC.EX ビットは非同期バウンスが発生した場合にのみ 1 にセットされます。

実装されている一般的な VFP サブアーキテクチャレジスタの検出 P. 付録 B-22 で説明されているように、ソフトウェアは FPEXC.EX および FPEXC.FP2V ビットをテストして、命令レジスタの存在を検出できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.3 Exception processing

The following sections describe exception processing in the Common VFP subarchitecture:

- *Asynchronous exceptions*
- *Synchronous exceptions* on page AppxB-8
- *VFP Access Permission faults* on page AppxB-10
- *Unallocated VFP instruction encodings* on page AppxB-10
- *Trapped floating-point exception handling* on page AppxB-10.

B.3.1 Asynchronous exceptions

In the Common VFP subarchitecture, an exceptional condition can be detected after executing the exceptional instruction. This means an implementation can detect an exceptional condition after an instruction has passed the point for exception handling in the ARM processor pipeline.

Handling this condition is known as asynchronous exception handling, because the exceptional condition can be detected some time after it is generated. In this case the exception handling:

- is signaled synchronously with respect to the trigger instruction
- is not signaled synchronously with respect to the instruction that generated the exceptional condition.

When an exceptional condition is detected the VFP coprocessor enters the asynchronous exceptional state, setting the FPEXC.EX bit to 1. At the application level, subsequent VFP instructions are rejected. This causes an Undefined Instruction exception, and information about the exceptional instruction is copied to:

- the FPINST register, see *The Floating-Point Instruction Registers, FPINST and FPINST2* on page AppxB-20
- the FPEXC.VECITR field.

For details of the FPEXC see:

- *The Floating-Point Exception Register (FPEXC)* on page B1-68 for the VFPv3 architectural requirements for the register
- *Additions to the Floating-Point Exception Register (FPEXC)* on page AppxB-15 for the Common VFP subarchitecture extensions to the register.

In some implementations it is possible for two VFP instructions to issue before an exceptional condition is detected in the first instruction. In this case the second instruction is copied to FPINST2, see *The Floating-Point Instruction Registers, FPINST and FPINST2* on page AppxB-20. This instruction must be executed by the support code. If there is a dependency between the instructions copied into FPINST and FPINST2 then the instruction in FPINST must be executed before the instruction in FPINST2.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B.3 例外処理

次に示すセクションでは、一般的な VFP サブアーキテクチャでの例外処理について説明します。

- *非同期例外*
- *同期例外* P. 付録 B-8
- *VFP アクセス許可フォールト* P. 付録 B-10
- *未割り当ての VFP 命令エンコード* P. 付録 B-10
- *トラップされた浮動小数点例外処理* P. 付録 B-10

B.3.1 非同期例外

一般的な VFP サブアーキテクチャでは、例外条件は例外命令の実行後に検出することができます。これは、命令が ARM プロセッサパイプライン内で例外処理用のポイントを通過した後に、実装が例外条件を検出できるということを意味します。

この条件の処理は、非同期例外処理と呼ばれます。これは、例外条件の検出が、生成から多少の時間が経過した後で可能になるためです。この場合、例外処理は以下のようになります。

- トリガ命令に同期して通知されます。
- 例外条件を生成した命令と同期して通知されません。

例外条件が検出されると、VFP コプロセッサは非同期例外状態に遷移し、FPEXC.EX ビットが 1 にセットされます。アプリケーションレベルでは、後続の VFP 命令が拒否されます。これは未定義命令例外を発生させ、例外命令に関する情報が次の場所にコピーされます。

- FPINST レジスタ（*浮動小数点命令レジスタ、FPINST および FPINST2* P. 付録 B-20 を参照）
- FPEXC.VECITR フィールド

FPEXC の詳細については、次に示すセクションを参照して下さい。

- *浮動小数点例外レジスタ (FPEXC)* P. B1-68（レジスタの VFPv3 アーキテクチャに関する要件）
- *浮動小数点例外レジスタ (FPEXC) への追加* P. 付録 B-15 （レジスタへの一般的な VFP サブアーキテクチャ拡張）

一部の実装では、最初の命令で例外条件が検出される前に、2 つの VFP 命令が発行されることがあります。この場合、2 番目の命令は FPINST2 にコピーされます。*浮動小数点命令レジスタ、FPINST および FPINST2* P. 付録 B-20 を参照して下さい。この命令は、サポートコードによって実行する必要があります。FPINST および FPINST2 にコピーされた命令の間に依存関係が存在する場合、FPINST 内の命令を FPINST2 内の命令より先に実行する必要があります。

トリガ命令は例外命令の直後の VFP 命令とは限らず、命令シーケンスによってはいくつもの命令の後でバウンスが発生することがあります。実装では、次のような場合、例外条件を検出する前にいくつかの VFP 命令を続けて実行することができます。

- これらの命令自体が例外ではない
- これらの命令が、例外命令に依存していない
- 例外命令のオペランドが、その命令の実行後もまだ使用可能である

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The trigger instruction might not be the VFP instruction immediately following the exceptional instruction, and depending on the instruction sequence, the bounce can occur many instructions later. An implementation can continue to execute some VFP instructions before detecting the exceptional condition, provided:

- these instructions are not themselves exceptional
- these instructions are independent of the exceptional instruction
- the operands for the exceptional instruction are still available after the execution of the instructions.

Determination of the trigger instruction

VMSR and VMRS instructions that access the FPEXC, FPSID, FPINST or FPINST2 registers do not trigger exception processing.

These system registers are not used in normal VFP application code, but are designed for use by support code and the operating system. Accesses to these registers do not bounce when the processor is in an asynchronous exceptional state, indicated by FPEXC.EX == 1. This means the support code can read information out of these registers, before clearing the exceptional condition by setting FPEXC.EX to 0.

All other VFP instructions, including VMSR and VMRS instructions that access the FPSCR, trigger exception processing if there is an outstanding exceptional condition. For more information, see *VFP support code* on page B1-70.

Exception processing for scalar instructions

When an exceptional condition is detected in a scalar CDP instruction:

- the exception-generating instruction is copied to the FPINST Register, see *The Floating-Point Instruction Registers, FPINST and FPINST2* on page AppxB-20
- the FPEXC.VECITR field is set to 0b111 to indicate that no short vector iterations are required
- the FPEXC.EX bit is set to 1
- all the operand registers to the instruction are restored to their original values, so that the instruction can be re-executed in support code
- If the execution of the instruction would set the cumulative exception flags for any exception, hardware might or might not set these flags.

  Note

  Because the cumulative exception flags are cumulative, it is always acceptable for the support code to set the exception flags to 1 as a result of emulating the instruction, even if the hardware has set them.

If there is a bypassed instruction then this is copied to the FPINST2 Register, and the FPEXC.FP2V bit is set to 1.

The next VFP instruction issued becomes the trigger instruction and causes entry to the operating system.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

トリガ命令の特定

FPEXC、FPSID、FPINST、または FPINST2 レジスタにアクセスする VMSR および VMRS 命令では、例外処理はトリガされません。

これらのシステムレジスタは通常の VFP アプリケーションコードでは使用されず、サポートコードおよびオペレーティングシステムが使用する目的で設計されています。プロセッサが FPEXC.EX == 1 で示される非同期例外状態の場合、これらのレジスタへのアクセスはバウンスされません。これは、FPEXC.EX を 0 に設定することにより例外条件をクリアする前に、サポートコードがこれらのレジスタから情報を読み出すことができることを意味します。

未解決の例外条件が存在する場合は、FPSCR にアクセスする VMSR および VMRS 命令を含む他のすべての VFP 命令により例外処理がトリガされます。詳細については、*VFP サポートコード* P. B1-70 を参照して下さい。

スカラ命令の例外処理

スカラ CDP 命令内で例外条件が検出された場合は、次の処理が実行されます。

- 例外生成命令が FPINST レジスタにコピーされます（*浮動小数点命令レジスタ、FPINST および FPINST2* P. 付録 B-20 を参照）。
- FPEXC.VECITR フィールドが 0b111 にセットされ、ショートベクタ繰り返しが必要ないことを示します。
- FPEXC.EX ビットが 1 にセットされます。
- 命令をサポートコードで再実行できるように、命令のすべてのオペランドレジスタに元の値を復元します。
- いずれかの例外に対して命令を実行することにより累積例外フラグがセットされる場合、ハードウェアがこれらのフラグをセットする場合もあり、セットしない場合もあります。

  注

  累積例外フラグは累積的であるため、ハードウェアがこれらのフラグをセットした場合でも、サポートコードが命令をエミュレートした結果として例外フラグを 1 にセットすることは常時可能です。

バイパスされた命令が存在する場合、この命令は FPINST2 レジスタにコピーされ、FPEXC.FP2V ビットが 1 にセットされます。

次に発行された VFP 命令はトリガ命令になり、オペレーティングシステムに制御が移ります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Exception processing for short vector instructions

With a short vector instruction, any iteration might be exceptional. When an exceptional condition is detected for a vector iteration, previous iterations can complete. For the exceptional iteration:

- The exception-generating instruction is copied to the FPINST register, see *The Floating-Point Instruction Registers, FPINST and FPINST2* on page AppxB-20. The source and destination registers are modified to point to the exceptional iteration.
- The FPEXC.VECITR field is written with the number of iterations remaining after the exceptional iteration.
- The FPEXC.EX bit is set to 1.
- The input operand registers to that iteration, and subsequent iterations, are restored to their original values.
- If the execution of the exception iteration, or subsequent iterations, would set the cumulative exception flags for any exception, hardware might or might not set these flags.

  Note

  Because the cumulative exception flags are cumulative, it is always acceptable for the support code to set the exception flags to 1 as a result of emulating the iterations of the instruction, even if the hardware has set them.

If there is a bypassed instruction then this is copied to the FPINST2 Register, and the FPEXC.FP2V bit is set to 1.

The next VFP instruction issued becomes the trigger instruction and causes entry to the operating system.

B.3.2 Synchronous exceptions

In the Common VFP subarchitecture, an implementation can signal a floating-point exception synchronously.

When an exceptional condition is detected in a CDP instruction, and the implementation chooses to signal the condition synchronously:

- if the exceptional condition is a trapped floating-point exception the FPEXC.DEX bit is set to 1
- if the reason for the exceptional condition is IMPLEMENTATION DEFINED then the value of the FPEXC.DEX bit is IMPLEMENTATION DEFINED
- the instruction is bounced, causing an Undefined Instruction exception
- FPEXC.EX is not set to 1.

The FPINST and FPINST2 registers are not used in this case.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ショートベクタ命令の例外処理

ショートベクタ命令では、任意の繰り返しで例外が発生する可能性があります。ベクタ繰り返しで例外条件が検出された場合、以前の繰り返しは完了できます。例外の発生した繰り返しについては、次の処理が実行されます。

- 例外生成命令が FPINST レジスタにコピーされます（*浮動小数点命令レジスタ、FPINST および FPINST2* P. 付録 B- 20 を参照）。ソースレジスタおよびデスティネーションレジスタが、例外の発生した繰り返しを指すように変更されます。
- FPEXC.VECITR フィールドに、例外の発生した繰り返しの後に残っている繰り返しの数が書き込まれます。
- FPEXC.EX ビットが 1 にセットされます。
- その繰り返しおよび後続の繰り返しの入力オペランドレジスタが、元の値に復元されます。
- いずれかの例外に対して、例外の発生した繰り返しまたは後続の繰り返しを実行することにより累積例外フラグがセットされる場合、ハードウェアがこれらのフラグをセットする場合もあり、セットしない場合もあります。

  注

  累積例外フラグは累積的であるため、ハードウェアがこれらのフラグをセットした場合でも、サポートコードが命令の繰り返しをエミュレートした結果として例外フラグを 1 にセットすることは常時可能です。

バイパスされた命令が存在する場合、この命令は FPINST2 レジスタにコピーされ、FPEXC.FP2V ビットが 1 にセットされます。

次に発行された VFP 命令はトリガ命令になり、オペレーティングシステムに制御が移ります。

B.3.2 同期例外

一般的な VFP サブアーキテクチャでは、実装により浮動小数点例外を同期して通知できます。

CDP 命令で例外条件が検出された場合、および実装により条件の同期した通知が選択されている場合は、次の処理が実行されます。

- 例外条件がトラップされた浮動小数点例外の場合、FPEXC.DEX ビットが 1 にセットされます。
- 例外条件の理由が実装定義である場合、FPEXC.DEX ビットの値は実装定義です。
- 命令がバウンスされ、未定義命令例外が発生します。
- FPEXC.EX は 1 にセットされません。

この場合、FPINST および FPINST2 レジスタは使用されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For scalar CDP instructions:

- All the operand registers to the instruction are restored to their original values, so that the instruction can be re-executed in support code.
- It is IMPLEMENTATION DEFINED whether the FPEXC.VV bit is set to 1. If it is, the FPEXC.VECITR field will contain 0b111.
- If the execution of the instruction would set the cumulative exception flags for any exception, hardware might or might not set these flags.

  Note

  Because the cumulative exception flags are cumulative, it is always acceptable for the support code to set the exception flags to 1 as a result of emulating the instruction, even if the hardware has set them.

For short vector instructions, any iteration might be exceptional. When an exceptional condition is detected for a vector iteration, previous iterations can complete. For the exceptional iteration:

- The FPEXC.VECITR field is written with a value that encodes the number of iterations remaining after the exceptional iteration. For details of the encoding see *Subarchitecture additions to the VFP system registers* on page AppxB-15.
- The FPEXC.VV bit is set to 1.
- The input operand registers to that iteration, and subsequent iterations, are restored to their original values.
- If the execution of the exception iteration, or subsequent iterations, would set the cumulative exception flags for any exception, hardware might or might not set these flags.

  Note

  Because the cumulative exception flags are cumulative, it is always acceptable for the support code to set the exception flags to 1 as a result of emulating the iterations of the instruction, even if the hardware has set them.

Note

- In version 1 of the Common VFP subarchitecture, all exceptions are signaled synchronously when the FPSCR.IXE bit is set to 1, see *Floating-point Status and Control Register (FPSCR)* on page A2-28. The FPEXC.DEX bit is RAZ/WI. For more information, see *Subarchitecture v1 exception handling when FPSCR.IXE == 1* on page AppxB-23.
- In version 2 of the Common VFP subarchitecture, exceptional conditions that cause synchronous exceptions are signaled by setting FPEXC.DEX to 1. For more information, see *Version 2 of the Common VFP subarchitecture* on page AppxB-24.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

スカラ CDP 命令については、次の処理が実行されます。

- 命令をサポートコードで再実行できるように、命令のすべてのオペランドレジスタが元の値に復元されます。
- FPEXC.VV ビットが 1 にセットされるかどうかは実装定義です。セットされる場合、FPEXC.VECITR フィールドは 0b111 になります。
- いずれかの例外について、命令の実行により累積例外フラグがセットされる場合、ハードウェアがこれらのフラグをセットする場合も、しない場合もあります。

  注

  累積例外フラグは累積的であるため、ハードウェアがこれらのフラグをセットした場合でも、サポートコードが命令をエミュレートした結果として例外フラグを 1 にセットすることは常時可能です。

ショートベクタ命令では、任意の繰り返しで例外が発生する可能性があります。ベクタ繰り返しで例外条件が検出された場合、以前の繰り返しは完了できます。例外の発生した繰り返しについては、次の処理が実行されます。

- FPEXC.VECITR フィールドに、例外の発生した繰り返しの後で残っている繰り返しの数をエンコードする値が書き込まれます。エンコードの詳細については、*VFP システムレジスタへのサブアーキテクチャ固有の追加* P. 付録 B-15 を参照して下さい。
- FPEXC.VV ビットが 1 にセットされます。
- その繰り返しおよび後続の繰り返しの入力オペランドレジスタが、元の値に復元されます。
- いずれかの例外に対して、例外の発生した繰り返しまたは後続の繰り返しを実行することにより累積例外フラグがセットされる場合、ハードウェアがこれらのフラグをセットする場合も、しない場合もあります。

  注

  累積例外フラグは累積的であるため、ハードウェアがこれらのフラグをセットした場合でも、サポートコードが命令の繰り返しをエミュレートした結果として例外フラグを 1 にセットすることは常時可能です。

注

- 一般的な VFP サブアーキテクチャのバージョン 1 では、FPSCR.IXE ビットが 1 にセットされた場合、すべての例外は同期して通知されます。*浮動小数点ステータスおよび制御レジスタ (FPSCR)* P. A2-28 を参照して下さい。FPEXC.DEX ビットは RAZ/WI です。詳細については、*FPSCR.IXE == 1 の場合のサブアーキテクチャ v1 の例外処理* P. 付録 B-23 を参照して下さい。
- 一般的な VFP サブアーキテクチャのバージョン 2 では、FPEXC.DEX を 1 にセットすることにより、同期例外を発生させる例外条件が通知されます。詳細については、*一般的な VFP サブアーキテクチャのバージョン 2* P. 付録 B-24 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.3.3 VFP Access Permission faults

When the VFP register bank is disabled by disabling coprocessors 10 and 11 in a coprocessor access control register, any attempt to use a VFP instruction will bounce. When the VFP register bank is disabled by clearing the FPEXC.EN bit to 0, any attempt to access a VFP registers, except the FPEXC or FPINST register, will bounce.

In a system where the VFP can be disabled, handler code must check that the VFP is enabled before processing a VFP exception.

B.3.4 Unallocated VFP instruction encodings

Unallocated VFP instruction encodings are those coprocessor 10 and 11 instruction encodings that are not allocated for VFP instructions by ARM.

An unallocated VFP instruction encoding bounces synchronously to the VFP Undefined Instruction handler code. In this case the VFP state is not modified, the FPEXC.EX bit is set to 0, and the FPEXC.DEX bit is set to 0. Unallocated instruction exception handling is synchronous.

The VFP exception handler code can check the FPEXC.EX bit, to find out if the VFP is using asynchronous exception handling to handle a previous exceptional condition.

If FPEXC.EX=1, the support code is called to process a previous exceptional instruction. On return from the support code the trigger instruction is reissued, and if the trigger instruction is an unallocated instruction the Undefined Instruction handler is re-entered, with FPEXC.EX=0.

If FPEXC.EN == 1, FPEXC.EX == 0 and FPEXC.DEX == 0, the handler code might have been called as a result of an unallocated instruction encoding or as a result of an allocated instruction encoding which has not been implemented:

- If the instruction is not a `CDP` instruction, the instruction is an unallocated instruction encoding and execution can jump to the unallocated instructions handler provided by the system.
- If the instruction is a `CDP` instruction, the support code must identify whether the instruction is one that it can handle. If it is not, then execution can jump to the unallocated instructions handler provided by the system.

B.3.5 Trapped floating-point exception handling

Trapped floating-point exceptions are never handled by hardware. When a trapped exception is detected by hardware the exception-generating instruction must be re-executed by the support code. The support code must re-detect and signal the exception.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B.3.3 VFP アクセス許可フォールト

コプロセッサのアクセス制御レジスタでコプロセッサ 10 および 11 を非稼動にすることにより VFP レジスタバンクが非稼動にされた場合、VFP 命令を使用する試みはすべてバウンスされます。FPEXC.EN ビットを 0 にクリアすることにより VFP レジスタバンクが非稼動にされた場合、FPEXC および FPINST レジスタ以外の VFP レジスタへのアクセスの試みはバウンスされます。

VFP を非稼動にできるシステムでは、VFP 例外を処理する前に、VFP が稼動していることをハンドラコードがチェックする必要があります。

B.3.4 未割り当ての VFP 命令エンコード

*未割り当ての VFP 命令エンコード*は、ARM により VFP 命令に対して割り当てられていないコプロセッサ 10 および 11 の命令エンコードです。

未割り当ての VFP 命令エンコードは、同期して VFP 未定義命令ハンドラコードにバウンスされます。この場合 VFP 状態は変更されず、FPEXC.EX ビットは 0 にクリアされ、FPEXC.DEX ビットも 0 にクリアされます。未割り当て命令例外処理は同期です。

VFP 例外ハンドラコードは、FPEXC.EX ビットをチェックして、VFP が非同期例外処理を使用して以前の例外条件を処理しているかどうかを判断します。

FPEXC.EX = 1 の場合、以前の例外命令を処理するためにサポートコードが呼び出されます。サポートコードからの復帰時にトリガ命令が再発行され、トリガ命令が未割り当て命令の場合は、FPEXC.EX = 0 で未定義命令ハンドラが再開されます。

FPEXC.EN == 1、FPEXC.EX == 0、かつ FPEXC.DEX == 0 の場合、未割り当て命令エンコードまたは実装されていない割り当て命令エンコードの結果として、ハンドラコードが呼び出されている可能性があります。

- 命令が `CDP` 命令ではない場合、その命令は未割り当て命令エンコードで、システムによって提供される未割り当て命令ハンドラに実行をジャンプさせることができます。
- 命令が `CDP` 命令である場合は、その命令が処理可能であるかどうかをサポートコードで識別する必要があります。そうでない場合は、システムによって提供される未割り当て命令ハンドラに実行をジャンプさせることができます。

B.3.5 トラップされた浮動小数点例外処理

トラップされた浮動小数点例外は、ハードウェアで処理されることはありません。トラップされた例外がハードウェアによって検出された場合は、サポートコードにより例外生成命令を再実行する必要があります。サポートコードは、例外を再度検出して通知する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.4 Support code requirements

When an instruction is bounced, control passes to the Undefined Instruction exception handler provided by the operating system.

The operating system is expected to:

1. Perform a standard exception entry sequence, preserving process state and re-enabling interrupts.
2. Decode the bounced instruction sufficiently to determine whether it is a coprocessor instruction, and if so, for which coprocessor.
3. Check whether the bounced instruction is conditional, and if it is conditional, check whether the condition was passed. This ensures correct execution on implementations that perform the bounce even for an instructions that would fail its condition code check.
4. Check whether the coprocessor is enabled in the access control register, and take appropriate action if not. For example, in the lazy context switch case described in *Context switching with the Advanced SIMD and VFP extensions* on page B1-69, the operating system context switches the VFP state.
5. Call an appropriate second-level handler for the coprocessor, passing in:
   - the instruction that bounced
   - the state of the associated process.
6. The second-level handler must indicate whether the bounced instruction is to be retried or skipped. It can also signal an additional exception that must be passed on to the application.
7. Restore the original process, transferring control to an exception handler in the application context if necessary.

If the bounced instruction is a VFP instruction, control is passed to a second-level handler for VFP coprocessor instructions. For the Common VFP subarchitecture this:

1. Uses the FPEXC.EX and FPEXC.DEX bits to determine the bounced instruction and associated handling. The three possible cases are:

   FPEXC.EX == 0, FPEXC.DEX == 0

   The bounce was synchronous. The exception-generating instruction is the instruction that bounced:

   - If the exception-generating instruction is not a `CDP` instruction, or the version of the subarchitecture is before version 3, the bounce was caused by an unallocated instruction encoding or a VFP access permission fault. Branch to operating system specific code that takes appropriate action.
   - If the exception-generating instruction is a `CDP` instruction, check whether the bounce was caused by a VFP access permission fault:
     — If it is a VFP access permission fault, branch to operating system specific code that takes appropriate action.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.4 サポートコードの要件

命令がバウンスされた場合、オペレーティングシステムによって提供される未定義命令例外ハンドラに制御が渡されます。

オペレーティングシステムは、次の処理を行うことが想定されます。

1. 標準の例外開始シーケンスを実行します。プロセス状態を保存し、割り込みを再許可します。
2. バウンスされた命令がコプロセッサの命令であるかどうかを判断するために、その命令を必要なだけデコードします。そうである場合は、対象のコプロセッサを特定します。
3. バウンスされた命令が条件付きであるかどうかをチェックします。条件付きの場合は、条件が満たされているかどうかをチェックします。これにより、条件コードチェックに失敗する命令についても、バウンスを実行する実装上で正確な実行が保証されます。
4. アクセス制御レジスタでコプロセッサが可能になっているかどうかをチェックします。そうでない場合は、適切な処置を実行します。例えば、*アドバンスト SIMD 拡張機能と VFP 拡張機能によるコンテキストスイッチ* P. B1-69 で説明されている低速コンテキストスイッチの場合、オペレーティングシステムのコンテキストが VFP 状態を切り替えます。
5. コプロセッサの適切な第 2 レベルのハンドラを呼び出し、次の情報を渡します。
   - バウンスされた命令
   - 関連するプロセスの状態
6. 第 2 レベルのハンドラは、バウンスされた命令を再試行するか、スキップするかを示す必要があります。また、アプリケーションに渡す必要がある追加の例外も通知することができます。
7. 元のプロセスを復元し、必要な場合は制御をアプリケーションコンテキストの例外ハンドラに移します。

バウンスされた命令が VFP 命令である場合、VFP コプロセッサ命令の第 2 レベルのハンドラに制御が渡されます。一般的な VFP サブアーキテクチャの場合は、これにより次の処理が実行されます。

1. FPEXC.EX および FPEXC.DEX ビットを使用して、バウンスされた命令と、それに関連する処理を決定します。以下の 3 つの可能性があります。

   FPEXC.EX == 0、FPEXC.DEX == 0

   バウンスは同期しています。例外生成命令は、次のようにバウンスされた命令です。

   - 例外生成命令が `CDP` 命令ではない場合、またはサブアーキテクチャのバージョンが 3 以前の場合、バウンスは未割り当て命令エンコードまたは VFP アクセス許可フォールトによって引き起こされたものです。適切な処置を実行する、オペレーティングシステム固有のコードに分岐します。
   - 例外生成命令が `CDP` 命令である場合は、バウンスが VFP アクセス許可フォールトによって引き起こされているかどうかを、次のようにチェックします。
     – VFP アクセス許可フォールトの場合は、適切な処置を実行するオペレーティングシステム固有のコードに分岐します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

— If it is a not a VFP access permission fault, determine the iteration count from FPSCR.LEN, and set the return address to the instruction following the bounced instruction. Then continue processing from step 2.

FPEXC.EX == 0, FPEXC.DEX == 1

The bounced instruction was executed as a valid floating-point operation, and it bounced because of an exceptional condition.

The exception-generating instruction is the instruction that bounced.

The iteration count is determined from either FPSCR.LEN or FPEXC.VECITR, depending on the value of FPEXC.VV:

- if FPEXC.VV is set to 0, the iteration count is determined from FPSCR.LEN
- if FPEXC.VV is set to 1, the iteration count is determined from FPEXC.VECITR.

Clear the FPEXC.DEX bit to 0, and set the return address to the instruction following the bounced instruction.

Continue processing from step 2.

FPEXC.EX == 1

The VFP bounce resulted from an asynchronous exception.

Collect information about the exceptional instruction, and any other instructions that are to be executed by support code. Clear the exceptional condition. For each instruction the data collected include the instruction encoding and the number of vector iterations.

This involves:

- Read the FPINST Register to find the exception-generating instruction.

  Read the FPEXC.VECITR field to find the remaining iteration count for this instruction.
- Check FPEXC.FP2V. If it is set to 1 there is a bypassed instruction:
  - — Read the FPINST2 Register to find the bypassed instruction
  - — Clear the FPEXC.EX and FPEXC.FP2V bits to 0.
  - — Read the FPSCR.LEN field to find the iteration count for the bypassed instruction.

    The FPSCR can be read-only when FPEXC.EX == 0.

  Otherwise there is no bypassed instruction:
  - — Clear FPEXC.EX to 0.

  FPEXC.EX == 0 indicates there is no subarchitecture state to context switch.

Set the return address to re-execute the trigger instruction.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

– VFP アクセス許可フォールトでない場合は、FPSCR.LEN から繰り返しカウントを決定し、復帰アドレスをバウンスされた命令の次の命令に設定します。次に、ステップ 2 から処理を続行します。

FPEXC.EX = 0、FPEXC.DEX = 1

バウンスされた命令は有効な浮動小数点演算として実行されており、例外条件が原因でバウンスされました。

例外生成命令は、バウンスされた命令です。

繰り返しカウントは、FPEXC.VV の値に応じて、FPSCR.LEN または FPEXC.VECITR のいずれかにより、次のように決定されます。

- FPEXC.VV が 0 にクリアされている場合、繰り返しカウントは FPSCR.LEN により決定されます。
- FPEXC.VV が 1 にセットされている場合、繰り返しカウントは FPEXC.VECITR により決定されます。

FPEXC.DEX ビットを 0 にクリアし、復帰アドレスをバウンスされた命令の次の命令に設定します。

ステップ 2 から処理を続行します。

FPEXC.EX = 1

VFP バウンスは、非同期例外の結果です。

例外命令、およびサポートコードにより実行される他のすべての命令に関する情報を収集します。例外条件をクリアします。それぞれの命令について収集されるデータには、命令エンコードおよびベクタ繰り返しの数が含まれます。

これには、次の操作が含まれます。

- FPINST レジスタを読み出して、例外生成命令を検出します。

  FPEXC.VECITR フィールドを読み出して、この命令の残りの繰り返しカウントを検出します。
- FPEXC.FP2V をチェックします。1 にセットされている場合、バイパスされた命令が存在します。
  - – FPINST2 レジスタを読み出し、バイパスされた命令を検出します。
  - – FPEXC.EX および FPEXC.FP2V ビットを 0 にクリアします。
  - – FPSCR.LEN フィールドを読み出し、バイパスされた命令の繰り返しカウントを検出します。

    FPSCR は、FPEXC.EX == 0 の場合は読み出し専用です。

  そうでない場合、バイパスされた命令は存在しません。
  - – FPEXC.EX を 0 にクリアします。

  FPEXC.EX == 0 は、コンテキストスイッチに対するサブアーキテクチャ状態が存在しないことを示します。

復帰アドレスを設定してトリガ命令を再実行します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

In version 1 of the Common VFP subarchitecture, the meaning of the FPEXC.EX bit changes when the FPSCR.IXE bit is set to 1. The FPSCR.IXE bit can be checked only after the FPEXC.EX bit is cleared to 0. If FPSCR.IXE is 0, go to step 2 below. If FPSCR.IXE is set to 1:

- the information collected from the VFP registers and the calculated return address are ignored
- the exception-generating instruction is the instruction that bounced, and the iteration count is the FPSCR.LEN value, as for the FPEXC.DEX == 1 case.
- set the return address to the instruction following the bounced instruction.

2. Packages up the information about the VFP instruction and iteration count into pairs in a form suitable to pass to the Computation Engine, described in step 3.

At this point the packaged information can be sent as a signal to another exception handler in the application, where the support code continues. Continuing in the application context makes it possible for the support code to call trap handlers directly, in the application.

3. Executes in software the instruction iterations described in step 2. All configuration information except vector length is read from the FPSCR.

   In previous support code implementations by ARM, this execution is performed by the VFP *Computation Engine* function.

   If trapped floating-point exceptions are enabled, the Computation Engine calls trap handlers as required.

   If the exceptional condition is an unallocated instruction, the Computation Engine will call a suitable error routine.

4. Returns to the appropriate return address.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

一般的な VFP サブアーキテクチャのバージョン 1 では、FPSCR.IXE ビットが 1 にセットされた場合に FPEXC.EX ビットの意味が変わります。FPSCR.IXE ビットは、FPEXC.EX ビットが 0 にクリアされた後にのみチェックすることができます。FPSCR.IXE が 0 の場合は、下の手順 2 に進みます。FPSCR.IXE が 1 にセットされている場合は、次の動作が行われます。

- VFP レジスタから収集された情報と、計算された復帰アドレスは無視されます。
- 例外生成命令はバウンスされた命令で、FPEXC.DEX == 1 の場合と同様に、繰り返しカウントは FPSCR.LEN の値です。
- 復帰アドレスを、バウンスされた命令の次の命令に設定します。

2. 手順 3 で説明されているように、VFP 命令および繰り返しカウントに関する情報を、計算エンジンに渡すのに適した形式のペアにパッケージ化します。

この時点で、パッケージ化された情報は、サポートコードが続行されるアプリケーション内の別の例外ハンドラへ信号として送信できます。アプリケーションコンテキスト内で続行することにより、サポートコードがアプリケーション内で直接トラップハンドラを呼び出すことが可能になります。

3. ソフトウェアで、手順 2 で説明されている命令繰り返しを実行します。ベクタ長以外のすべての構成情報が、FPSCR から読み出されます。

   ARM による以前のサポートコードでは、これは VFP の*計算エンジン*機能によって実行されます。

   トラップされた浮動小数点例外が可能な場合、計算エンジンは必要に応じてトラップハンドラを呼び出します。

   例外条件が未割り当て命令の場合、計算エンジンは該当するエラールーチンを呼び出します。

4. 適切な復帰アドレスに戻ります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.5 Context switching

Context switch code must check the FPEXC.EX bit when saving or restoring VFP state.

If the FPEXC.EX bit is set to 1 then additional subarchitecture information must be saved. Any attempt to access other registers while the FPEXC.EX bit is set to 1 might bounce.

For the Common VFP subarchitecture, if the FPEXC.EX bit is set to 1:

- the FPINST register contains a bounced instruction and must be saved
- if the FPEXC.FP2V bit is set, the FPINST2 register must be saved.

The FPEXC register must always be saved.

When the subarchitecture specific information has been saved, context switch code must clear the FPEXC.EX bit to 0 before saving other registers.

When restoring state, check the saved values of the FPEXC.EX bit and FPEXC.FP2V bit to determine whether the extra registers must be restored.

Note

Context switch code can be written to always save and restore the subarchitecture registers. In this case appropriate context switch code must be chosen based on the registers implemented, using the detection mechanism described in *Detecting which VFP Common subarchitecture registers are implemented* on page AppxB-22.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B.5 コンテキストスイッチ

コンテキストスイッチ コードは、VFP 状態が保存または復元されたときに FPEXC.EX ビットをチェックする必要があります。

FPEXC.EX ビットが 1 にセットされている場合は、追加のサブアーキテクチャ情報を保存する必要があります。FPEXC.EX ビットが 1 にセットされている間に他のレジスタにアクセスを試みると、バウンスが発生する可能性があります。

一般的な VFP サブアーキテクチャでは、FPEXC.EX ビットが 1 にセットされている場合には次の規則が適用されます。

- FPINST レジスタにはバウンスされた命令が格納されており、保存が必要です。
- FPEXC.FP2V ビットがセットされている場合は、FPINST2 レジスタの保存が必要です。

FPEXC レジスタは、常に保存が必要です。

サブアーキテクチャ固有の情報が保存された場合、コンテキストスイッチ コードでは、他のレジスタを保存する前に FPEXC.EX ビットを 0 にクリアする必要があります。

状態を復元するときは、FPEXC.EX ビットおよび FPEXC.FP2V ビットの保存されている値をチェックし、別のレジスタを復元する必要があるかどうかを判断します。

注

コンテキストスイッチ コードは、常にサブアーキテクチャレジスタの保存および復元を行うように記述してもかまいません。この場合は、*実装されている一般的な VFP サブアーキテクチャレジスタの検出* P. 付録 B- 22 で説明されている検出機構を使用し、実装されているレジスタに基づいて適切なコンテキストスイッチ コードを選択する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.6 Subarchitecture additions to the VFP system registers

The Common VFP subarchitecture requires additions to the VFP system register implementation:

- extra fields are defined in the FPEXC, see *Additions to the Floating-Point Exception Register (FPEXC)*
- additional VFP registers might be defined, see *The Floating-Point Instruction Registers, FPINST and FPINST2* on page AppxB-20.

Also, the Subarchitecture field of the FPSID must identify the Common VFP subarchitecture version, see *Floating-point System ID Register (FPSID)* on page B5-34.

For more information about the VFP register implementation for the Common VFP subarchitecture see:

- *Detecting which VFP Common subarchitecture registers are implemented* on page AppxB-22
- *Accessing the VFP Common subarchitecture registers* on page AppxB-22.

B.6.1 Additions to the Floating-Point Exception Register (FPEXC)

See *The Floating-Point Exception Register (FPEXC)* on page B1-68 for the architectural definition of the FPEXC, including its purpose and accessibility.

The format of the FPEXC when version 3 of the Common VFP subarchitecture is implemented is:

EX, bit [31] See *The Floating-Point Exception Register (FPEXC)* on page B1-68 for the definition of this bit.

On an implementation that does not require asynchronous exception handling this bit is RAZ/WI. In this case the FPINST and FPINST2 registers are not implemented.

For details of how, in Common VFP subarchitecture v1, the meaning of the EX bit changes when the FPSR.IEX bit is set to 1, see *Subarchitecture v1 exception handling when FPSCR.IXE == 1* on page AppxB-23.

EN, bit [30] See *The Floating-Point Exception Register (FPEXC)* on page B1-68 for the definition of this bit.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.6 VFP システムレジスタへのサブアーキテクチャ固有の追加

一般的な VFP サブアーキテクチャでは、VFP システムレジスタ実装への追加が必要です。

- 追加フィールドは FPEXC で定義されます（*浮動小数点例外レジスタ (FPEXC) への追加*を参照）。
- 追加の VFP レジスタが定義される場合があります（*浮動小数点命令レジスタ、FPINST および FPINST2* P. 付録 B-20 を参照）。

また、FPSID のサブアーキテクチャフィールドは、一般的な VFP サブアーキテクチャのバージョンを識別する必要があります。*浮動小数点システム ID レジスタ (FPSID)* P. B5-34 を参照して下さい。

一般的な VFP サブアーキテクチャの VFP レジスタ実装に関する詳細については、次に示すセクションを参照して下さい。

- *実装されている一般的な VFP サブアーキテクチャレジスタの検出* P. 付録 B-22
- *一般的な VFP サブアーキテクチャレジスタへのアクセス* P. 付録 B-22

B.6.1 浮動小数点例外レジスタ (FPEXC) への追加

FPEXC の目的やアクセス可能性など、アーキテクチャの定義については、*浮動小数点例外レジスタ (FPEXC)* P. B1-68 を参照して下さい。

一般的な VFP サブアーキテクチャのバージョン 3 が実装されている場合、FPEXC のフォーマットは次のとおりです。

EX、ビット [31] このビットの定義については、*浮動小数点例外レジスタ (FPEXC)* P. B1-68 を参照して下さい。

非同期例外処理を必要としない実装では、このビットは RAZ/WI です。この場合、FPINST および FPINST2 レジスタは実装されません。

一般的な VFP サブアーキテクチャ v1 において、FPSR.IEX ビットが 1 にセットされた場合に EX ビットの意味がどのように変更されるかの詳細については、*FPSCR.IXE == 1 の場合のサブアーキテクチャ v1 の例外処理* P. 付録 B-23 を参照して下さい。

EN、ビット [30] このビットの定義については、*浮動小数点例外レジスタ (FPEXC)* P. B1-68 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

DEX, bit [29] Defined synchronous instruction exceptional flag. This field is valid only if FPEXC.EX == 0.

When a VFP exception has occurred, the meaning of this bit is:

0 A synchronous exception has occurred when processing an instruction in CP10 or CP 11. This is either:

- an allocated floating-point instruction that is not implemented in hardware
- an unallocated instruction in CP10 or CP11.

1 A synchronous exception has occurred on an allocated floating-point instruction. This is either:

- an allocated floating-point instruction that is not implemented in hardware
- an allocated floating-point instruction that has encountered an exceptional condition.

DEX must be cleared to 0 by the exception handling routine.

On an implementation that does not require synchronous exception handling this bit is RAZ/WI.

FP2V, bit [28] FPINST2 instruction valid flag. This field is valid only if FPEXC.EX == 1.

When an asynchronous VFP exception has occurred, the meaning of this bit is:

0 The FPINST2 Register does not contain a valid instruction.

1 The FPINST2 Register contains a valid instruction.

FP2V must be cleared to 0 by the exception handling routine.

If the FPINST2 Register is not implemented this bit is RAZ/WI.

VV, bit [27] VECITR valid flag. This field is valid only if FPEXC.DEX == 1.

When a synchronous VFP exception has occurred, the meaning of this bit is:

0 FPEXC.VECITR field is not valid, and the number of remaining vector steps can be determined from FPSCR.LEN.

1 FPEXC.VECITR field is valid, and the number of remaining vector steps can be determined from FPEXC.VECITR.

VV must be cleared to 0 by the exception handling routine.

If the VV field s not implemented this bit is RAZ/WI.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

DEX、ビット [29]

定義済み同期命令例外フラグ。このフィールドは、FPEXC.EX == 0 の場合にのみ有効です。

VFP 例外が発生した場合、このビットは次の意味を持ちます。

0 CP10 または CP11 での命令処理で同期例外が発生しました。これは、次のいずれかです。

- ハードウェアに実装されていない割り当て済み浮動小数点命令
- CP10 または CP11 内の未割り当て命令

1 割り当て済み浮動小数点命令で、同期例外が発生しました。これは、次のいずれかです。

- ハードウェアに実装されていない割り当て済み浮動小数点命令
- 例外条件が発生した割り当て済み浮動小数点命令

DEX は、例外処理ルーチンで 0 にクリアする必要があります。

同期例外処理を必要としない実装では、このビットは RAZ/WI です。

FP2V、ビット [28]

FPINST2 命令有効フラグ。このフィールドは、FPEXC.EX == 1 の場合にのみ有効です。

非同期 VFP 例外が発生した場合、このビットの意味は次のとおりです。

0 FPINST2 レジスタには有効な命令が格納されていません。

1 FPINST2 レジスタには有効な命令が格納されています。

FP2V は、例外処理ルーチンで 0 にクリアする必要があります。

FPINST2 レジスタが実装されていない場合、このビットは RAZ/WI です。

VV、ビット [27]

VECITR 有効フラグ。このフィールドは、FPEXC.DEX = 1 の場合にのみ有効です。

同期 VFP 例外が発生した場合、このビットの意味は次のとおりです。

0 FPEXC.VECITR フィールドは有効ではありません。残りのベクタステップの数は FPSCR.LEN から決定できます。

1 FPEXC.VECITR フィールドは有効で、残りのベクタステップの数は FPEXC.VECITR から決定できます。

VV は、例外処理ルーチンで 0 にクリアする必要があります。

VV フィールドが実装されていない場合、このビットは RAZ/WI です。

TFV、ビット [26]

トラップされたフォールトフラグ有効フラグ。FPEXC のビット [7、4:0] がトラップされた例外を示すフラグとして動作することを示すか、または次のような実装定義の意味を持ちます。

0 FPEXC のビット [7、4:0] は、実装定義の意味を持ちます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

TFV, bit [26] Trapped Fault flag Valid flag. Indicates whether FPEXC bits [7,4:0] act as flags to indicate trapped exceptions or have an IMPLEMENTATION DEFINED meaning:

0 FPEXC bits[7,4:0] have an IMPLEMENTATION DEFINED meaning

1 FPEXC bits[7,4:0] indicate the presence of trapped exceptions that have occurred at the time of the exception. All trapped exceptions that occurred at the time of the exception have their flags set.

This bit has a fixed value and ignores writes.

Bits [25:21] Reserved. UNK/SBZP.

Bits [20:11, 6:5]

IMPLEMENTATION DEFINED.

These bits are IMPLEMENTATION DEFINED. They can contain IMPLEMENTATION DEFINED information about the cause of an exception. They might be used by the implementation to indicate why an instruction was bounced to support code.

These bits must be cleared to zero by the exception handling routine.

VECITR, bits [10:8]

Vector iteration count for the VFP instruction with the exceptional condition. This field is valid only if either:

- FPEXC.EX == 1
- FPEXC.DEX == 1 and FPEXC.VV == 1.

This field contains the number of short vector iterations remaining after the iteration in which a potential exception was detected. Possible values are:

0b000 1 iteration
0b001 2 iterations
0b010 3 iterations
0b011 4 iterations
0b100 5 iterations
0b101 6 iterations
0b110 7 iterations
0b111 0 iterations.

The count held in this field does not include the iteration in which the exception occurred. This field reads as 0b111 if:

- the final iteration of an instruction is bounced to the support code
- the instruction is a scalar operation.

VECITR must be cleared to 0b000 by the exception handling routine.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

1 FPEXC のビット [7、4:0] は、例外の時点で発生したトラップされた例外が存在することを示します。例外の時点で発生したすべてのトラップされた例外は、フラグがセットされます。

このビットの値は固定されており、書き込みは無視されます。

ビット [25:21] 予約。UNK/SBZP。

ビット [20:11、6:5]

実装定義。

これらのビットは実装定義です。例外の原因に関する実装定義の情報を格納できます。これらのビットは、命令がサポートコードにバウンスされた理由を示すために実装により使用される場合があります。

これらのビットは、例外処理ルーチンで 0 にクリアする必要があります。

VECITR、ビット [10:8]

例外条件での VFP 命令のベクタ繰り返しカウント。このフィールドは、次のいずれかの場合にのみ有効です。

- FPEXC.EX == 1
- FPEXC.DEX == 1 かつ FPEXC.VV == 1

このフィールドには、潜在的な例外が検出された繰り返し後に残っているショートベクタ繰り返しの数が格納されます。可能な値は次のとおりです。

0b000 1 回の繰り返し
0b001 2 回の繰り返し
0b010 3 回の繰り返し
0b011 4 回の繰り返し
0b100 5 回の繰り返し
0b101 6 回の繰り返し
0b110 7 回の繰り返し
0b111 0 回の繰り返し

このフィールドに保持されているカウントには、例外が発生した繰り返しは含まれていません。次の場合、このフィールドの読み出し値は 0b111 です。

- 命令の最後の繰り返しがサポートコードにバウンスされた場合
- 命令がスカラ演算である場合

VECITR は、例外処理ルーチンで 0b000 にクリアする必要があります。

IDF、ビット [7]

入力非正規化トラップ例外フラグ、または実装定義。このビットの意味は、FPEXC.TFV の値によって次のように異なります。

FPEXC.TFV == 0

このビットは実装定義です。例外の原因に関する実装定義の情報が含まれている可能性があります。このビットは、命令がサポートコードにバウンスされた理由を示すために実装により使用される場合があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

IDF, bit [7] Input Denormal trapped exception flag, or IMPLEMENTATION DEFINED. The meaning of this bit depends on the value of FPEXC.TFV:

FPEXC.TFV == 0

This bit is IMPLEMENTATION DEFINED. It can contain IMPLEMENTATION DEFINED information about the cause of an exception. It might be used by the implementation to indicate why an instruction was bounced to support code.

FPEXC.TFV == 1

This bit is the Input Denormal trapped exception flag. It indicates whether an Input Denormal exception occurred while FPSCR.IDE was 1.

In this case, the meaning of this bit is:

0 Input denormal exception has not occurred.

1 Input denormal exception has occurred.

Input Denormal exceptions can occur only when FPSCR.FZ is 1.

In both cases this bit must be cleared to 0 by the exception handling routine.

IXF, bit [4] Inexact trapped exception flag, or IMPLEMENTATION DEFINED. The meaning of this bit depends on the value of FPEXC.TFV:

FPEXC.TFV == 0

This bit is IMPLEMENTATION DEFINED. It can contain IMPLEMENTATION DEFINED information about the cause of an exception. It might be used by the implementation to indicate why an instruction was bounced to support code.

FPEXC.TFV == 1

This bit is the Inexact trapped exception flag. It indicates whether an Inexact exception occurred while FPSCR.IXE was 1.

In this case, the meaning of this bit is:

0 Inexact exception has not occurred.

1 Inexact exception has occurred.

In both cases this bit must be cleared to 0 by the exception handling routine.

UFF, bit [3] Underflow trapped exception flag, or IMPLEMENTATION DEFINED. The meaning of this bit depends on the value of FPEXC.TFV:

FPEXC.TFV == 0

This bit is IMPLEMENTATION DEFINED. It can contain IMPLEMENTATION DEFINED information about the cause of an exception. It might be used by the implementation to indicate why an instruction was bounced to support code.

FPEXC.TFV == 1

This bit is the Underflow trapped exception flag. It indicates whether an Underflow exception occurred while FPSCR.UFE was 1.

In this case, the meaning of this bit is:

0 Underflow exception has not occurred.

1 Underflow exception has occurred.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

FPEXC.TFV = 1

このビットは、入力非正規化トラップ例外フラグです。FPSCR.IDE が 1 の間に、入力非正規化例外が発生したかどうかを示します。

この場合、このビットの意味は次のとおりです。

0 . 入力非正規化例外は発生していません。

1 . 入力非正規化例外が発生しました。

入力非正規化例外は、FPSCR.FZ が 1 の場合にのみ発生します。

いずれの場合も、このビットは例外処理ルーチンで 0 にクリアする必要があります。

IXF、ビット [4]

不正確トラップ例外フラグ、または実装定義。このビットの意味は、FPEXC.TFV の値によって次のように異なります。

FPEXC.TFV = 0

このビットは実装定義です。例外の原因に関する実装定義の情報が含まれている可能性があります。このビットは、命令がサポートコードにバウンスされた理由を示すために実装により使用される場合があります。

FPEXC.TFV = 1

このビットは、不正確トラップ例外フラグです。FPSCR.IXE ビットが 1 の間に不正確例外が発生したかどうかを示します。

この場合、このビットの意味は次のとおりです。

0 不正確例外は発生していません。

1 不正確例外が発生しました。

いずれの場合も、このビットは例外処理ルーチンで 0 にクリアする必要があります。

UFF、ビット [3]

アンダーフロートラップ例外フラグ、または実装定義。このビットの意味は、FPEXC.TFV の値によって次のように異なります。

FPEXC.TFV = 0

このビットは実装定義です。例外の原因に関する実装定義の情報が含まれている可能性があります。このビットは、命令がサポートコードにバウンスされた理由を示すために実装により使用される場合があります。

FPEXC.TFV = 1

このビットは、アンダーフロートラップ例外フラグです。FPSCR.UFE が 1 の間にアンダーフロー例外が発生したかどうかを示します。

この場合、このビットの意味は次のとおりです。

0 アンダーフロー例外は発生していません。

1 アンダーフロー例外が発生しました。

注

アンダーフロートラップ例外は、FPSCR.FZ が 0 のときにのみ発生します。FPSCR.FZ が 1 の場合、FPSCR.UFE は無視され 0 として処理されるためです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

An Underflow trapped exception can occur only when FPSCR.FZ is 0, because when FPSCR.FZ is 1, FPSCR.UFE is ignored and treated as 0.

In both cases this bit must be cleared to 0 by the exception handling routine.

OFF, bit [2] Overflow trapped exception flag, or IMPLEMENTATION DEFINED. The meaning of this bit depends on the value of FPEXC.TFV:

FPEXC.TFV == 0

This bit is IMPLEMENTATION DEFINED. It can contain IMPLEMENTATION DEFINED information about the cause of an exception. It might be used by the implementation to indicate why an instruction was bounced to support code.

FPEXC.TFV == 1

This bit is the Overflow trapped exception flag. It indicates whether an Overflow exception occurred while FPSCR.OFE was 1.

In this case, the meaning of this bit is:

0 Overflow exception has not occurred.

1 Overflow exception has occurred.

In both cases this bit must be cleared to 0 by the exception handling routine.

DZF, bit [1] Divide-by-zero trapped exception flag, or IMPLEMENTATION DEFINED. The meaning of this bit depends on the value of FPEXC.TFV:

FPEXC.TFV == 0

This bit is IMPLEMENTATION DEFINED. It can contain IMPLEMENTATION DEFINED information about the cause of an exception. It might be used by the implementation to indicate why an instruction was bounced to support code.

FPEXC.TFV == 1

This bit is the Divide-by-zero trapped exception flag. It indicates whether a Divide-by-zero exception occurred while FPSCR.DZE was 1.

In this case, the meaning of this bit is:

0 Divide-by-zero exception has not occurred.

1 Divide-by-zero exception has occurred.

In both cases this bit must be cleared to 0 by the exception handling routine.

IOF, bit [0] Invalid Operation trapped exception flag, or IMPLEMENTATION DEFINED. The meaning of this bit depends on the value of FPEXC.TFV:

FPEXC.TFV == 0

This bit is IMPLEMENTATION DEFINED. It can contain IMPLEMENTATION DEFINED information about the cause of an exception. It might be used by the implementation to indicate why an instruction was bounced to support code.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

いずれの場合も、このビットは例外処理ルーチンで 0 にクリアする必要があります。

OFF、ビット [2]

オーバフロートラップ例外フラグ、または実装定義。このビットの意味は、FPEXC.TFV の値によって次のように異なります。

FPEXC.TFV == 0

このビットは実装定義です。例外の原因に関する実装定義の情報が含まれている可能性があります。このビットは、命令がサポートコードにバウンスされた理由を示すために実装により使用される場合があります。

FPEXC.TFV == 1

このビットは、オーバフロートラップ例外フラグです。FPSCR.OFE が 1 の間にオーバフロー例外が発生したかどうかを示します。

この場合、このビットの意味は次のとおりです。

0 オーバフロー例外は発生していません。

1 オーバフロー例外が発生しました。

いずれの場合も、このビットは例外処理ルーチンで 0 にクリアする必要があります。

DZF、ビット [1]

0 による除算トラップ例外フラグ、または実装定義。このビットの意味は、FPEXC.TFV の値によって次のように異なります。

FPEXC.TFV == 0

このビットは実装定義です。例外の原因に関する実装定義の情報が含まれている可能性があります。このビットは、命令がサポートコードにバウンスされた理由を示すために実装により使用される場合があります。

FPEXC.TFV == 1

このビットは、0 による除算トラップ例外フラグです。FPSCR.DZE ビットが 1 の間に 0 による除算例外が発生したかどうかを示します。

この場合、このビットの意味は次のとおりです。

0 0 による除算例外は発生していません。

1 0 による除算例外が発生しました。

いずれの場合も、このビットは例外処理ルーチンで 0 にクリアする必要があります。

IOF、ビット [0]

無効演算トラップ例外フラグ、または実装定義。このビットの意味は、FPEXC.TFV の値によって次のように異なります。

FPEXC.TFV == 0

このビットは実装定義です。例外の原因に関する実装定義の情報が含まれている可能性があります。このビットは、命令がサポートコードにバウンスされた理由を示すために実装により使用される場合があります。

FPEXC.TFV == 1

このビットは、無効演算トラップ例外フラグです。FPSCR.IOE が 1 の間に無効演算例外が発生したかどうかを示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

FPEXC.TFV == 1

This bit is the Invalid Operation trapped exception flag. It indicates whether an Invalid Operation exception occurred while FPSCR.IOE was 1.

In this case, the meaning of this bit is:

0 Invalid Operation exception has not occurred.

1 Invalid Operation exception has occurred.

In both cases this bit must be cleared to 0 by the exception handling routine.

B.6.2 The Floating-Point Instruction Registers, FPINST and FPINST2

The Floating-Point Instruction Registers hold floating-point instructions relating to floating-point exception handling in a system that implements the Common VFP subarchitecture:

- FPINST contains the exception-generating instruction
- FPINST2 contains the bypassed instruction.

FPINST and FPINST2 are:

- In the CP10 and CP11 register space.
- Present only when the Common VFP subarchitecture is implemented. A Common VFP subarchitecture implementation can support:
  - — both FPINST and FPINST2
  - — FPINST but not FPINST2
  - — neither of the Floating-Point Instruction Registers.
- 32-bit read/write registers.
- If the Security Extensions are implemented, Configurable access registers. FPINST and FPINST2 are only accessible in the Non-secure state if the CP10 and CP11 bits in the NSACR are set to 1, see *c1, Non-Secure Access Control Register (NSACR)* on page B3-110.
- Accessible only in privileged modes, and only if both:
  - — access to coprocessors CP10 and CP11 is enabled in the Coprocessor Access Control Register, see *c1, Coprocessor Access Control Register (CPACR)* on page B3-104 (VMSA implementation), or *c1, Coprocessor Access Control Register (CPACR)* on page B4-51 (PMSA implementation)
  - — the VFP coprocessor is enabled by setting the FPEXC.EN bit to 1.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

この場合、このビットの意味は次のとおりです。

0 無効演算例外は発生していません。

1 無効演算例外が発生しました。

いずれの場合も、このビットは例外処理ルーチンで 0 にクリアする必要があります。

B.6.2 浮動小数点命令レジスタ、FPINST および FPINST2

浮動小数点命令レジスタには、一般的な VFP サブアーキテクチャが実装されているシステムでの浮動小数点例外処理に関連する浮動小数点命令が保持されます。

- FPINST には、例外生成命令が格納されます。
- FPINST2 には、バイパスされた命令が格納されます。

FPINST および FPINST2 には、次の特徴があります。

- CP10 および CP11 のレジスタ空間に存在します。
- 一般的な VFP サブアーキテクチャが実装されている場合にのみ存在します。一般的な VFP サブアーキテクチャ実装では、次のいずれかをサポートできます。
  - – FPINST と FPINST2 の両方
  - – FPINST はサポートするが FPINST2 はサポートしない
  - – 浮動小数点命令レジスタのいずれもサポートしない
- 32 ビットの読み出し / 書き込みレジスタ
- セキュリティ拡張機能が実装されている場合、構成可能なアクセスレジスタです。FPINST および FPINST2 は、NSACR の CP10 および CP11 ビットが 1 にセットされている場合、非セキュア状態においてのみアクセス可能です。*c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3- 110 を参照して下さい。
- 特権モードで、次の両方の条件が満たされている場合のみアクセス可能です。
  - – コプロセッサアクセス制御レジスタでコプロセッサ CP10 および CP11 へのアクセスが可能になっている。*c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B3- 104（VMSA 実装の場合）、または *c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B4- 51（PMSA 実装の場合）を参照して下さい。
  - – FPEXC.EN ビットが 1 にセットされ、VFP コプロセッサが稼動している。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of an instruction in FPINST or FPINST2 is:

Thumb encoding

15							8	7	6	5	4	3	2	1	0	15	14	13	12	11			8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0		D			Vn				Vd				cp_num				N	Q	M	0	Vm			

ARM encoding

31							24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0		D			Vn				Vd				cp_num				N	Q	M	0	Vm			

The format is the same as the format of the issued instruction, with a number of modifications. For more information, see *VFP data-processing instructions* on page A7-24. The modifications from the issued instruction are:

- In the Thumb encoding, bits [15:8] of the first halfword and bit [4] of the second halfword are reserved. In the ARM encoding, bits [31:24, 4] are reserved:
  - — software must ignore these bits when reading this register, and must not modify these bits when writing to this register
  - — hardware must set these bits to the values shown in the encoding diagrams, that map to the encoding of an ARM `CDP` instruction with the AL (always) condition.
- If the instruction is a short vector instruction:
  - — for the FPINST Register, the source and destination registers that reference vectors are updated to point to the source and destination registers of the exceptional iteration. The FPEXC.VECITR field contains the number of iterations remaining. For more information, see *Exception processing for short vector instructions* on page AppxB-8.
  - — for the FPINST Register, the full vector must be processed by support code, using the current vector length from the FPSCR. Source and destination registers that reference vectors are unchanged from the issued instruction.

Both `MRS` register read and `MSR` register write instructions are provided for the FPINST and FPINST2 registers, see *Accessing the VFP Common subarchitecture registers* on page AppxB-22.

When an exceptional instruction is bounced to support code and placed in the FPINST Register, the FPEXC.EX bit is set to 1. This indicates that valid information is available in the FPINST Register. In addition. when a second issued instruction is copied to the FPINST2 Register, the FPEXC.FP2V bit is set to 1. This indicates that valid information is available in the FPINST2 Register.

When the FPEXC.EX bit is 0, indicating the VFP is not in an asynchronous exceptional state, reads of the FPINST and FPINST2 Registers are UNPREDICTABLE and the values returned might change.

When the FPEXC.FP2V bit is 0, indicating that no second instruction was issued, reads of the FPINST2 Register are UNPREDICTABLE and the value returned might change.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

FPINST または FPINST2 での命令のフォーマットは、次のとおりです。

Thumb エンコード

15							8	7	6	5	4	3	2	1	0	15	14	13	12	11			8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0		D			Vn				Vd				cp_num				N	Q	M	0	Vm			

ARM エンコード

31							24	23	22	21	20	19	18	17	16	15	14	13	12	11			8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0		D			Vn				Vd				cp_num				N	Q	M	0	Vm			

このフォーマットは発行された命令のフォーマットと同じですが、多数の変更点があります。詳細については、*VFP のデータ処理命令* P. A7-24 を参照して下さい。発行された命令からの変更点は、次のとおりです。

- Thumb エンコードでは、最初のハーフワードのビット [15:8] および 2 番目のハーフワードのビット [4] は予約されています。ARM エンコードでは、ビット [31:24、4] は予約されています。
  - – ソフトウェアは、このレジスタの読み出しの際にはこれらのビットを無視し、書き込みの際にはこれらのビットを変更しないようにする必要があります。
  - – ハードウェアは、これらのビットをエンコード図で示される値に設定する必要があります。これにより、AL（常時）条件として ARM の `CDP` 命令のエンコードにマップされます。
- 命令がショートベクタ命令の場合、次の規則が適用されます。
  - – FPINST レジスタの場合、ベクタを参照しているソースレジスタとデスティネーションレジスタが、例外の発生した繰り返しのソースレジスタとデスティネーションレジスタを指すように更新されます。FPEXC.VECITR フィールドには、残りの繰り返し回数が格納されます。詳細については、*ショートベクタ命令の例外処理* P. 付録 B-8 を参照して下さい。
  - – FPINST レジスタの場合、FPSCR の現在のベクタ長を使用して、サポートコードにより完全なベクタを処理する必要があります。ベクタを参照しているソースレジスタおよびデスティネーションレジスタは、発行された命令から変更されません。

FPINST および FPINST2 レジスタでは、`MRS` レジスタ読み出し命令と `MSR` レジスタ書き込み命令の両方が提供されます。*一般的な VFP サブアーキテクチャレジスタへのアクセス* P. 付録 B-22 を参照して下さい。

例外命令がサポートコードにバウンスされ、FPINST レジスタに置かれると、FPEXC.EX ビットが 1 にセットされます。これは、FPINST レジスタに有効な情報が存在することを示します。さらに、2 番目に発行された命令が FPINST2 レジスタにコピーされると、FPEXC.FP2V ビットが 1 にセットされます。これは、FPINST2 レジスタに有効な情報が存在することを示します。

FPEXC.EX ビットが 0 の場合、VFP が非同期例外状態ではないことを示します。この場合、FPINST および FPINST2 レジスタの読み出し値は予測不能で、返される値は変化する可能性があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Any value read from a Floating-Point Instruction Register can be written back to the same register. This means context switch and debugger software can save and restore Floating-Point Instruction Register values. Writing a value that has not been read from the same register writes an UNKNOWN value to the Floating-Point Instruction Register. For example, attempting to write an instruction with coprocessor number 0 writes an UNKNOWN value to the Floating-Point Instruction Register.

B.6.3 Accessing the VFP Common subarchitecture registers

Use the VMRS and VMSR instructions to access the registers for the VFP Common subarchitecture implementation, see:

- *VMRS* on page B6-27
- *VMSR* on page B6-29.

The additional registers in the VFP Common subarchitecture are accessed using:

- reg = 0b1001 for FPINST
- reg = 0b1010 for FPINST2

If FPINST or FPINST2 is not defined, the corresponding VMRS and VMSR instructions are UNPREDICTABLE.

The VMRS and VMSR instructions with reg = 0b1011 and reg = 0b11xx are UNPREDICTABLE.

B.6.4 Detecting which VFP Common subarchitecture registers are implemented

An implementation can choose not to implement FPINST and FPINST2, if these registers are not required.

System software can detect which registers are present as follows:

```
Set FPEXC.EX=1 and FPEXC.FP2V=1
Read back the FPEXC register
if FPEXC.EX == 0 then
    Neither FPINST nor FPINST2 are implemented
else
    if FPEXC.FP2V == 0 then
        FPINST is implemented, FPINST2 is not implemented.
    else
        Both FPINST and FPINST2 are implemented.
Clean up
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

FPEXC.FP2V ビットが 0 の場合、2 番目の命令が発行されなかったことを示します。この場合、FPINST2 レジスタの読み出し値は予測不能で、返される値は変化する可能性があります。

浮動小数点命令レジスタから読み出された値はすべて、同じレジスタに書き戻すことができます。これは、コンテキストスイッチおよびデバッガソフトウェアが、浮動小数点命令レジスタの値を保存して復元できることを意味します。同じレジスタから読み出されたものではない値を書き込むと、未知の値が浮動小数点命令レジスタに書き込まれます。例えば、コプロセッサ番号 0 を指定して命令の書き込みを試みると、未知の値が浮動小数点命令レジスタに書き込まれます。

B.6.3 一般的な VFP サブアーキテクチャレジスタへのアクセス

一般的な VFP サブアーキテクチャ実装のレジスタにアクセスするには、VMRS および VMSR 命令を使用します。次に示すセクションを参照して下さい。

- *VMRS* P. B6-27
- *VMSR* P. B6-29

一般的な VFP サブアーキテクチャの追加レジスタにアクセスするには、次のパラメータを使用します。

- reg = 0b1001（FPINST の場合）
- reg = 0b1010（FPINST2 の場合）

FPINST または FPINST2 が定義されていない場合、対応する VMRS および VMSR 命令は予測不能です。

VMRS および VMSR 命令で reg = 0b1011 および reg = 0b11xx の場合は予測不能です。

B.6.4 実装されている一般的な VFP サブアーキテクチャレジスタの検出

実装では、FPINST および FPINST2 が必要ない場合、これらのレジスタを実装しないことを選択できます。

システムソフトウェアは、次の手順を使用して、存在するレジスタを検出できます。

```
Set FPEXC.EX=1 and FPEXC.FP2V=1
Read back the FPEXC register
if FPEXC.EX == 0 then
    Neither FPINST nor FPINST2 are implemented
else
    if FPEXC.FP2V == 0 then
        FPINST is implemented, FPINST2 is not implemented
    else
        Both FPINST and FPINST2 are implemented
Clean up
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.7 Version 1 of the Common VFP subarchitecture

Version 1 of the Common VFP subarchitecture has special behavior when the FPSCR.IXE bit is set to 1. The Common VFP subarchitecture version can be identified by checking FPSID bits [22:16]. This field is 0b0000001 for version 1. In version 1 of the Common VFP subarchitecture the FPEXC.DEX bit is RAZ/WI.

B.7.1 Subarchitecture v1 exception handling when FPSCR.IXE == 1

In version 1 of the Common VFP subarchitecture, the mechanism for bouncing instructions changes when the FPSCR.IXE bit, the Inexact exception enable bit, is set to 1.

When FPSCR.IXE is set to 1, the FPEXC.EX bit signals a synchronous exception, in the same way as the FPEXC.DEX bit. In this case:

- the exceptional instruction is the instruction that caused the Undefined Instruction exception
- the FPINST Register and the FPEXC.VECITR field are not valid.

When FPSCR.IXE is 0 the FPEXC.EX bit signals an asynchronous exception, as for later versions of the subarchitecture.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.7 一般的な VFP サブアーキテクチャのバージョン 1

一般的な VFP サブアーキテクチャのバージョン 1 は、FPSCR.IXE ビットが 1 にセットされている場合、特別な動作をします。一般的な VFP サブアーキテクチャのバージョンは、FPSID のビット [22:16] をチェックして識別できます。バージョン 1 では、このフィールドは 0b0000001 です。一般的な VFP サブアーキテクチャのバージョン 1 では、FPEXC.DEX ビットは RAZ/WI です。

B.7.1 FPSCR.IXE ＝ 1 の場合のサブアーキテクチャ v1 の例外処理

一般的な VFP サブアーキテクチャのバージョン 1 では、命令をバウンスする機構は、FPSCR.IXE ビット（不正確例外イネーブルビット）が 1 にセットされた場合に変更されます。

FPSCR.IXE が 1 にセットされている場合、FPEXC.DEX ビットと同様に、FPEXC.EX ビットにより同期例外が通知されます。この場合、次の規則が適用されます。

- 例外命令は、未定義命令例外を引き起こした命令です。
- FPINST レジスタおよび FPEXC.VECITR フィールドは無効です。

FPSCR.IXE が 0 の場合、サブアーキテクチャの以降のバージョンと同様に、FPEXC.EX ビットにより非同期例外が通知されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

B.8 Version 2 of the Common VFP subarchitecture

Version 2 of the Common VFP subarchitecture can be identified by checking FPSID bits [22:16]. This field is 0b0000010 for version 2.

Version 2 of the Common VFP subarchitecture has three differences from version 3 of the subarchitecture. Before version 3 of the Common VFP subarchitecture:

- The FPEXC.EX == 0, FPEXC.DEX == 0 encoding is used only for unallocated instructions or permission faults. As a result, the determination that an instruction should be passed to the Computation Engine is simpler than it is for version 3 of the Common VFP subarchitecture.
- Bounces are not handled synchronously on short vector instructions unless all iterations of the vector are to be handled in software. This means that the FPEXC.VV bit is always 0 before version 3.
- The FPEXC.TFV bit is set to 0, so the additional information bits [7, 4:0] of FPEXC is IMPLEMENTATION DEFINED.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

B.8 一般的な VFP サブアーキテクチャのバージョン 2

一般的な VFP サブアーキテクチャのバージョン 2 は、FPSID のビット [22:16] をチェックして識別できます。バージョン 2 では、このフィールドは 0b0000010 です。

一般的な VFP サブアーキテクチャのバージョン 2 には、サブアーキテクチャのバージョン 3 と異なる点が 3 つあります。一般的な VFP サブアーキテクチャのバージョン 3 以前では、次の規則が適用されます。

- FPEXC.EX == 0、FPEXC.DEX == 0 であるエンコードは、未割り当て命令またはアクセス許可フォールトに対してのみ使用されます。結果として、命令を計算エンジンに渡す必要性の判断は、一般的な VFP サブアーキテクチャのバージョン 3 よりも単純です。
- ベクタのすべての繰り返しがソフトウェア内で処理されない限り、ショートベクタ命令でバウンスは同期して処理されません。これは、バージョン 3 以前では FPEXC.VV ビットが常時 0 であることを意味します。
- FPEXC.TFV ビットが 0 にクリアされるため、FPEXC の追加情報ビット [7、4:0] は実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Appendix C
Legacy Instruction Mnemonics

This appendix provides information about the Unified Assembler Language equivalents of older assembler language instruction mnemonics.

It contains the following sections:

- *Thumb instruction mnemonics* on page AppxC-2
- *Pre-UAL pseudo-instruction NOP* on page AppxC-3.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

付録 C
過去の命令のニーモニック

この付録では、以前のアセンブラ言語命令ニーモニックと等価の統一アセンブラについて解説します。

本章は次のセクションから構成されています。

- *Thumb 命令ニーモニック* P. 付録 C- 2
- *UAL 以前の擬似命令 NOP* P. 付録 C- 3

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C.1 Thumb instruction mnemonics

Table C-1 lists the UAL equivalents of the mnemonics used in pre-UAL Thumb assembly language. Except where noted, the Thumb mnemonics conflict with UAL and cannot be supported by assemblers as synonyms. Thumb code cannot be correctly assembled by a UAL assembler unless these changes are made.

All other Thumb instructions are the same in UAL as in Thumb assembler language, or can be supported as synonyms.

Table C-1 Thumb instruction mnemonics

Former Thumb assembler mnemonic	UAL equivalent
ADC	ADCS
ADD	ADDS [a]
AND	ANDS
ASR	ASRS
BIC	BICS
EOR	EORS
LSL	LSLS
MOV <Rd>,#<imm>	MOVS <Rd>,#<imm>
MOV <Rd>,<Rn>	ADDS <Rd>,<Rn>,#0 [b]
MUL	MULS
MVN	MVNS
ORR	ORRS
ROR	RORS
SBC	SBCS
SUB	SUBS [c]

a. If either or both of the operands is R8-R15, ADD not ADDS.
b. If either or both of the operands is R8-R15, MOV <Rd>,<Rn> not ADDS <Rd>,<Rn>,#0.
c. If the operand register is SP, SUB not SUBS.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C.1 Thumb 命令ニーモニック

UAL 以前の Thumb アセンブラ言語で使用されているニーモニックと等価の UAL を、表 C-1 に示します。特に記載されている場合を除いて、Thumb ニーモニックは UAL と競合し、アセンブラで同義語としてサポートすることはできません。Thumb コードは、これらの変更を加えない限り UAL アセンブラで正しくアセンブルできません。

他のすべての Thumb 命令は、Thumb アセンブラ言語と UAL とで同じであるか、または同義語としてサポートすることができます。

表 C-1 Thumb 命令のニーモニック

以前の Thumb アセンブラのニーモニック	等価な UAL の命令
ADC	ADCS
ADD	ADDS [a]
AND	ANDS
ASR	ASRS
BIC	BICS
EOR	EORS
LSL	LSLS
MOV <Rd>,#<imm>	MOVS <Rd>,#<imm>
MOV <Rd>,<Rn>	ADDS <Rd>,<Rn>,#0 [b]
MUL	MULS
MVN	MVNS
ORR	ORRS
ROR	RORS
SBC	SBCS
SUB	SUBS [c]

a. オペランドのいずれかまたは両方が R8 ～ R15 の場合は、ADDS ではなく ADD を使用します。
b. オペランドのいずれかまたは両方が R8 ～ R15 の場合は、ADDS <Rd>,<Rn>,#0 ではなく MOV <Rd>,<Rn> を使用します。
c. オペランドレジスタが SP の場合は、SUBS ではなく SUB を使用します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C.2 Pre-UAL pseudo-instruction NOP

In pre-UAL assembler code, `NOP` is a pseudo-instruction, equivalent to:

- `MOV R0,R0` in ARM code
- `MOV R8,R8` in Thumb code.

Assembling the `NOP` mnemonic as UAL will not change the functionality of the code, but will change:

- the instruction encoding selected
- the architecture variants on which the resulting binary will execute successfully, because the `NOP` instruction was introduced in ARMv6K and ARMv6T2.

To avoid these changes, replace `NOP` in the assembler source code with the appropriate one of `MOV R0,R0` and `MOV R8,R8`, before assembling as UAL.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

C.2 UAL 以前の擬似命令 NOP

UAL 以前のアセンブラコードでは、`NOP` は擬似命令で、次の命令と等価です。

- `MOV R0,R0`（ARM コードの場合）
- `MOV R8,R8`（Thumb コードの場合）

`NOP` ニーモニックを UAL としてアセンブルしても、コードの機能は変更されません。ただし、次の内容は変更されます。

- 選択した命令エンコード
- 結果のバイナリが正常に実行されるアーキテクチャバリアント。これは、`NOP` 命令が ARMv6K および ARMv6T2 で導入されているためです。

これらの変更を回避するには、UAL としてアセンブルする前に、アセンブラ ソースコードの `NOP` を `MOV R0,R0` または `MOV R8,R8` の適切な方の命令で置き換えます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Appendix D
Deprecated and Obsolete Features

This appendix contains the following sections:

- *Deprecated features* on page AppxD-2
- *Deprecated terminology* on page AppxD-5
- *Obsolete features* on page AppxD-6
- *Semaphore instructions* on page AppxD-7
- *Use of the SP as a general-purpose register* on page AppxD-8
- *Explicit use of the PC in ARM instructions* on page AppxD-9
- *Deprecated Thumb instructions* on page AppxD-10.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

付録 D
非推奨または廃止された機能

この付録は次のセクションから構成されています。

- *非推奨の機能* P. 付録 D-2
- *非推奨の用語* P. 付録 D-5
- *廃止された機能* P. 付録 D-6
- *セマフォ命令* P. 付録 D-7
- *SP の汎用レジスタとしての使用* P. 付録 D-8
- *ARM 命令での PC の明示的な使用* P. 付録 D-9
- *非推奨の Thumb 命令* P. 付録 D-10

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.1 Deprecated features

The features described in this section are present in ARMv7 for backwards compatibility. You must avoid using them in new applications where possible. They might not be present in future versions of the ARM architecture.

See also *Semaphore instructions* on page AppxD-7, *Use of the SP as a general-purpose register* on page AppxD-8, and *Explicit use of the PC in ARM instructions* on page AppxD-9.

D.1.1 VFP vector mode

The use of VFP vector mode is deprecated in ARMv7. For details see Appendix F *VFP Vector Operation Support*.

D.1.2 VFP FLDMX and FSTMX instructions

The use of `VLDM.64` and `VSTM.64` instruction encodings with an odd immediate offset is deprecated from ARMv6. The use of their pre-UAL mnemonics `FLDMX` and `FSTMX` is deprecated, except for disassembly purposes. For details see *FLDMX, FSTMX* on page A8-101.

D.1.3 Fast context switch extension

Use of the *Fast Context Switch Extension* (FCSE) is deprecated from ARMv6, and in ARMv7 implementation of the FCSE is optional. For details of the FCSE see Appendix E *Fast Context Switch Extension (FCSE)*.

D.1.4 Direct manipulation of the Endianness bit

The use of the `MSR` instruction to write the Endianness bit in User mode is deprecated. Use the `SETEND` instruction.

D.1.5 Strongly-ordered memory accesses and interrupt masks

Any ARMv5 instruction that implicitly or explicitly changes the interrupt masks in the CPSR, and appears in program order after a Strongly-ordered access, waits for the Strongly-ordered memory access to complete. Dependence on this behavior is deprecated in ARMv6 and ARMv7, and code must not rely on this behavior. Use an explicit memory barrier instead. For details see *Strongly-ordered memory* on page A3-34.

D.1.6 Unaligned exception returns

ARM deprecates any dependence on the requirements that the hardware ignores bits of the address transferred to the PC on an exception return. See *Alignment of exception returns* on page B1-39.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

D.1 非推奨の機能

このセクションで説明されている機能は、下位互換性のため ARMv7 に搭載されています。可能な限り、新規アプリケーションでこれらの機能を使用することは避ける必要があります。これらの機能は、ARM アーキテクチャの今後のバージョンには搭載されない可能性があります。

セマフォ命令 P. 付録 D-7、*SP の汎用レジスタとしての使用* P. 付録 D-8、*ARM 命令での PC の明示的な使用* P. 付録 D-9 も参照して下さい。

D.1.1 VFP ベクタモード

VFP ベクタモードの使用は、ARMv7 では推奨されません。詳細については、付録 F *「VFP ベクタ演算のサポート」*を参照して下さい。

D.1.2 VFP の FLDMX および FSTMX 命令

奇数イミディエートオフセットでの `VLDM.64` および `VSTM.64` 命令エンコードの使用、ARMv6 以降では推奨されません。これに相当する UAL 以前のニーモニック `FLDMX` および `FSTMX` の使用は、逆アセンブリ目的の場合を除いて推奨されません。詳細については、*FLDMX*、*FSTMX* P. A8-101 を参照して下さい。

D.1.3 高速コンテキストスイッチ拡張機能

高速コンテキストスイッチ拡張機能 (FCSE) の使用は ARMv6 およびそれ以降では非推奨で、ARMv7 では FCSE の実装はオプションです。FCSE の詳細については、付録 E *「高速コンテキストスイッチ拡張機能 (FCSE)」*を参照して下さい。

D.1.4 エンディアン形式ビットの直接操作

ユーザモードで `MSR` 命令を使用してエンディアン形式ビットを書き込むことは推奨されません。`SETEND` 命令を使用して下さい。

D.1.5 ストロングリオーダ メモリアクセスと割り込みマスク

CPSR の割り込みマスクを暗黙的または明示的に変更し、プログラムの順序でストロングリオーダアクセスの後に出現する ARMv5 命令は、ストロングリオーダ メモリアクセスの完了を待ちます。この動作における依存性は ARMv6 および ARMv7 では推奨されておらず、コードはこの動作に依存することはできません。代わりに、明示的メモリバリアを使用して下さい。詳細については、*ストロングリオーダ メモリ* P. A3-34 を参照して下さい。

D.1.6 アンアラインドな例外からの復帰

例外からの復帰の際に PC に転送されるアドレスのビットをハードウェアが無視する要件に関する依存性は推奨されません。*例外からの復帰のアライメント* P. B1-39 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.1.7 Use of AP[2] = 1, AP[1:0] = 0b10 in MMU access permissions

This encoding means read-only for both privileged mode and User mode accesses, but its use is deprecated in VMSAv7. Use AP[2] = 1, AP[1:0] = 0b11. For details see *Memory access control* on page B3-28.

D.1.8 The Domain field in the DFSR

Use of the Domain field in the DFSR is deprecated. For details see *c6, Data Fault Address Register (DFAR)* on page B3-124.

D.1.9 Watchpoint Fault Address Register in CP15

Use of the CP15 alias of the *Watchpoint Fault Address Register* (DBGWFAR) is deprecated. Use the CP14 DBGWFAR instead. For details see *Extended CP14 interface* on page C6-33.

D.1.10 CP15 memory barrier operations

Use of the CP15 c7 memory barrier operations is deprecated. The ARM and Thumb instruction sets include instructions that perform these operations. Table D-1 shows the deprecated CP15 encodings and the replacement ARMv7 instructions.

Table D-1 Deprecated CP15 c7 memory barrier operations

Deprecated CP15 encoding				Operation	Instruction description
CRn	opc1	CRm	opc2		
c7	0	c5	4	Instruction Synchronization Barrier	See *ISB* on page A8-102
c7	0	c10	4	Data Synchronization Barrier	See *DSB* on page A8-92
c7	0	c10	5	Data Memory Barrier	See *DMB* on page A8-90

D.1.11 Use of Hivecs exception base address in PMSA implementations

Use of the high vector exception base address (Hivecs) of `0xFFFF0000` is deprecated in PMSA implementations. ARM recommends that Hivecs is used only in VMSA implementations. For more information, see *Exception vectors and the exception base address* on page B1-30.

D.1.12 Use of Secure User halting debug

From v7 Debug, the use of Secure User halting debug is deprecated. For more information, see *About invasive debug authentication* on page C2-2.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.1.7 MMU アクセス許可での AP[2] = 1、AP[1:0] = 0b10 の使用

このエンコードは、特権モードおよびユーザモードの両方のアクセスについて読み出し専用であることを意味しますが、VMSAv7 ではこれを使用することは推奨されません。AP[2] = 1、AP[1:0] = 0b11 を使用して下さい。詳細については、*メモリアクセス制御* P. B3-28 を参照して下さい。

D.1.8 DFSR のドメインフィールド

DFSR のドメインフィールドを使用することは推奨されません。詳細については、*c6、データ フォールト アドレスレジスタ (DFAR)* P. B3-124 を参照して下さい。

D.1.9 CP15 でのウォッチポイントフォールトアドレス レジスタ

ウォッチポイントフォールトアドレス レジスタ (DBGWFAR) の CP15 エイリアスを使用することは推奨されません。代わりに、CP14 DBGWFAR を使用して下さい。詳細については、*拡張 CP14 インタフェース* P. C6-33 を参照して下さい。

D.1.10 CP15 のメモリバリア操作

CP15 c7 のメモリバリア操作の使用は推奨されません。ARM および Thumb 命令セットには、これらの操作を実行する命令が含まれています。非推奨の CP15 エンコードおよびその置き換えとなる ARMv7 命令を、表 D-1 に示します。

表 D-1 非推奨の CP15 c7 メモリバリア操作

非推奨の CP15 エンコード				動作	命令の説明
CRn	opc1	CRm	opc2		
c7	0	c5	4	命令同期バリア	*ISB* P. A8-102 を参照
c7	0	c10	4	データ同期バリア	*DSB* P. A8-92 を参照
c7	0	c10	5	データ メモリバリア	*DMB* P. A8-90 を参照

D.1.11 PMSA 実装における Hivecs 例外ベースアドレスの使用

PMSA 実装において、ハイベクタ例外ベースアドレス (Hivecs) である `0xFFFF0000` を使用することは推奨されません。VMSA 実装においてのみ Hivecs を使用することをお勧めします。詳細については、*例外ベクタと例外ベースアドレス* P. B1-30 を参照して下さい。

D.1.12 セキュアなユーザ ホールトデバッグの使用

v7 デバッグおよびそれ以降では、セキュアなユーザ ホールトデバッグの使用は推奨されません。詳細については、*侵襲性デバッグの認証について* P. C2-2 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.1.13 Escalation of privilege on CP14 and CP15 accesses in Debug state

Except for the Baseline CP14 debug registers, ARM deprecates accessing any CP14 or CP15 register from User mode in Debug state if that register cannot be accessed from User mode in Non-debug state. For more information, see *Coprocessor and Advanced SIMD instructions in Debug state* on page C5-16.

D.1.14 Interrupts or asynchronous aborts in a sequence of memory transactions

ARM deprecates any reliance by software on the behavior that an interrupt or asynchronous abort cannot occur in a sequence of single-copy atomic memory transactions generated by a single load/store instruction to Normal memory. For more information, see *Low interrupt latency configuration* on page B1-43.

D.1.15 Reading the Debug Program Counter Sampling Registers as register 33

ARM deprecates reading the DBGPCSR as debug register 33 when it is also implemented as debug register 40. For more information see *Program Counter sampling* on page C8-2.

D.1.16 Old mnemonics for CP15 c8 operations to invalidate entries in a unified TLB

The ARMv7-A base architecture defines three CP15 c8 operations to invalidate entries in a unified TLB. The original mnemonics for these are changed, each dropping the initial U. The original mnemonics remain synonyms for the operations, but ARM deprecates using the old mnemonics. Table D-2 shows the changed mnemonics and the encodings of the operations.

Table D-2 Changed mnemonics for CP15 c8 unified TLB operations

Encoding				Mnemonic	
CRn	opc1	CRm	opc2	New	Deprecated
c8	0	c7	0	TLBIALL	UTLBIALL
			1	TLBIMVA	UTLBIMVA
			2	TLBIMVA	UTLBIMVA

For more information about these operations see *CP15 c8, TLB maintenance operations* on page B3-138.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

D.1.13 デバッグ状態における CP14 および CP15 アクセスの特権の昇格

ベースライン CP14 デバッグレジスタを除いて、非デバッグ状態のユーザモードから CP14 または CP15 レジスタにアクセスできない場合、デバッグ状態のユーザモードからそれらのレジスタにアクセスすることは推奨されません。詳細については、*デバッグ状態でのコプロセッサおよびアドバンスト SIMD 命令* P. C5-16 を参照して下さい。

D.1.14 メモリトランザクションのシーケンスにおける割り込みまたは非同期アボート

ノーマルメモリへの単一のロード / ストア命令によって生成されるシングルコピーアトミックのメモリトランザクションのシーケンスにおいて、割り込みまたは非同期アボートが発生しない動作にソフトウェアが依存することは推奨されません。詳細については、*低割り込みレイテンシ構成* P. B1-43 を参照して下さい。

D.1.15 デバッグプログラムカウンタサンプリング レジスタのレジスタ 33 としての読み出し

DBGPCSR がデバッグレジスタ 40 としても実装されている場合、デバッグレジスタ 33 として読み出すことは推奨されません。詳細については、*プログラムカウンタのサンプリング* P. C8-2 を参照して下さい。

D.1.16 統一 TLB のエントリを無効化する CP15 c8 操作の以前のニーモニック

ARMv7-A 基本アーキテクチャには、統一 TLB のエントリを無効化する 3 つの CP15 c8 演算が定義されています。これらの元のニーモニックは変更されており、それぞれ頭文字 U が抜けています。元のニーモニックは操作の同義語のままですが、以前のニーモニックの使用は推奨されません。変更後のニーモニックおよびエンコードを表 D-2 に示します。

表 D-2 CP15 c8 統一 TLB 操作の変更後のニーモニック

エンコード				ニーモニック	
CRn	opc1	CRm	opc2	新規	非推奨
c8	0	c7	0	TLBIALL	UTLBIALL
			1	TLBIMVA	UTLBIMVA
			2	TLBIMVA	UTLBIMVA

これらの操作の詳細については、*CP15 c8*、*TLB 保守操作* P. B3-138 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.2 Deprecated terminology

Table D-3 shows terms that were used in earlier editions of the *ARM Architecture Reference Manual*, and the supplements to it, that are no longer used. The replacement terms are not in general exact synonyms, but might reflect altered behavior more accurately.

Table D-3 Deprecated terminology

Old terminology	Replaced by
Drain Write Buffer, Data Write Barrier (DWB)	Data Synchronization Barrier
Prefetch Flush (PFF)	Instruction Synchronization Barrier

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.2 非推奨の用語

『*ARM アーキテクチャ リファレンスマニュアル*』の以前のエディションでは使用されていたが現在は使用されていない用語、およびその置き換えとなる用語を表 D-3 に示します。置き換えとなる用語は一般に正確な同義語ではありませんが、変更された動作をより正確に反映している場合があります。

表 D-3 非推奨の用語

以前の用語	置き換えとなる用語
ライトバッファドレイン、データライトバリア (DWB)	データ同期バリア
プリフェッチフラッシュ (PFF)	命令同期バリア

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.3 Obsolete features

The features described in the following sections were deprecated in ARMv6, and are no longer supported in ARMv7.

D.3.1 Rotated aligned accesses

Unaligned accesses, where permitted, were treated as rotated aligned accesses before ARMv6. This behavior was configurable, but deprecated, in ARMv6. It is obsolete in ARMv7. For more information, see *Alignment* on page AppxG-6.

D.3.2 Ordering of instructions that change the CPSR interrupt masks

Any ARMv6 instruction that implicitly or explicitly changes the interrupt masks in the CPSR and appears in program order after a Strongly-ordered access must wait for the Strongly-ordered memory access to complete, see *Ordering of instructions that change the CPSR interrupt masks* on page AppxG-8 for more information.

ARMv6 deprecated any reliance on this behavior, and this behavior is obsoleted in ARMv7.

D.3.3 ARM LDM and POP instructions that both write back and load their base registers

LDM instructions and multi-register POP instructions that specify base register writeback and load their base register are permitted but deprecated before ARMv7, as described in *Different definition of some LDM and POP instructions* on page AppxG-15. Use of such instructions is obsolete in ARMv7.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

D.3 廃止された機能

以下のセクションで説明されている機能は ARMv6 では非推奨で、ARMv7 ではサポートされていません。

D.3.1 ローテート アラインドアクセス

アンアラインドアクセスが許可されている場合、ARMv6 以前ではローテート アラインドアクセスとして扱われます。この動作は、ARMv6 では構成可能ですが非推奨です。ARMv7 では廃止されています。詳細については、*アライメント* P. 付録 G-6 を参照して下さい。

D.3.2 CPSR 割り込みマスクを変更する命令の順序付け

CPSR で暗黙的または明示的に割り込みマスクを変更し、プログラムの順序でストロングリオーダアクセスの後に出現する ARMv6 命令は、ストロングリオーダ メモリアクセスの完了を待つ必要があります。詳細については、*CPSR 割り込みマスクを変更する命令の順序付け* P. 付録 G-8 を参照して下さい。

ARMv6 ではこの動作への依存は非推奨で、この動作は ARMv7 では廃止されています。

D.3.3 ベースレジスタのライトバックおよびロードの両方を行う ARM の LDM および POP 命令

一部の LDM および POP 命令の別の定義 P. 付録 G-15 で説明されているように、ベースレジスタ ライトバックを指定してそのベースレジスタのロードを行う LDM 命令およびマルチレジスタ POP 命令は、ARMv7 以前では許可されますが、非推奨です。このような命令の使用は、ARMv7 では廃止されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.4 Semaphore instructions

The ARM instruction set has two semaphore instructions:

- Swap (SWP)
- Swap Byte (SWPB).

These instructions are provided for process synchronization. Both instructions generate a load access and a store access to the same memory location, such that no other access to that location is permitted between the load access and the store access. This enables a memory semaphore to be loaded and altered without interruption.

SWP and SWPB have a single addressing mode, whose address is the contents of a register. Separate registers are used to specify the value to store and the destination of the load. If the same register is specified for both of these, SWP exchanges the value in the register and the value in memory.

The semaphore instructions do not provide a compare and conditional write facility. If wanted, this must be done explicitly.

Use of the swap and swap byte instructions is deprecated from ARMv6. ARM recommends that all software uses the LDREX and STREX synchronization primitives. For details see:

- *LDREX* on page A8-142
- *LDREXB* on page A8-144
- *LDREXD* on page A8-146
- *LDREXH* on page A8-148
- *STREX* on page A8-400
- *STREXB* on page A8-402
- *STREXD* on page A8-404
- *STREXH* on page A8-406.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.4 セマフォ命令

ARM 命令セットには、次の 2 つのセマフォ命令があります。

- スワップ (SWP)
- バイトスワップ (SWPB)

これらの命令は、プロセスの同期用に提供されているものです。どちらの命令でも、同じメモリ位置へのロードアクセスおよびストアアクセスが生成され、ロードアクセスとストアアクセスとの間でその位置へ他のアクセスは許可されません。これにより、割り込みなしにメモリセマフォがロードされて変更されることが可能になります。

SWP および SWPB のアドレシングモードは単一で、レジスタの内容がアドレスとして使用されます。ストアする値およびロードのデスティネーションの指定には、別のレジスタが使用されます。これらの両方に同じレジスタが指定されている場合、SWP によりレジスタの値とメモリの値が交換されます。

セマフォ命令では、比較および条件付き書き込み機能は提供されません。必要な場合は、明示的に実行する必要があります。

スワップおよびバイトスワップ命令の使用は、ARMv6 およびそれ以降では非推奨です。すべてのソフトウェアで、LDREX および STREX 同期化基本命令を使用することをお勧めします。詳細については、次の項目を参照して下さい。

- *LDREX* P. A8- 142
- *LDREXB* P. A8- 144
- *LDREXD* P. A8- 146
- *LDREXH* P. A8- 148
- *STREX* P. A8- 400
- *STREXB* P. A8- 402
- *STREXD* P. A8- 404
- *STREXH* P. A8- 406

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.5 Use of the SP as a general-purpose register

In the Thumb instruction set, you can only use the SP (R13) in a restricted set of instructions. This set covers all the legitimate uses of the SP as a stack pointer. An attempt to encode any other instruction with SP in place of a legitimate register results in either UNPREDICTABLE behavior, or a different instruction.

In addition, the use of SP (R13) as Rm in the high register forms of the 16-bit CMP and ADD instructions is deprecated. Also, some forms of MOV (register) that use SP are deprecated, see *MOV (register)* on page A8-196.

Most ARM instructions, unlike Thumb instructions, provide exactly the same access to the SP as to R0-R12. This means that it is possible to use the SP as a general-purpose register. However, the use of the SP in an ARM instruction, in any way that is not possible in the corresponding Thumb instruction, is deprecated.

See *ARM instructions where SP use is not deprecated* for a list of instructions that you can use for SP manipulation.

D.5.1 ARM instructions where SP use is not deprecated

The use of the SP is deprecated in any ARM instruction that is not specified in this section.

Some uses of the SP are not deprecated in the following ARM data-processing instructions:

- *ADD (SP plus immediate)* on page A8-28
- *ADD (SP plus register)* on page A8-30
- *CMN (immediate)* on page A8-74
- *CMN (register)* on page A8-76
- *CMP (immediate)* on page A8-80
- *CMP (register)* on page A8-82
- *MOV (register)* on page A8-196
- *SUB (SP minus immediate)* on page A8-426
- *SUB (SP minus register)* on page A8-428.

In these ARM instructions, the uses of the SP that are not deprecated are the same as those uses listed in *32-bit Thumb instruction support for R13* on page A6-4.

The use of the SP as the base register in load/store/preload instructions is not deprecated. In addition, the use of the SP as destination or source register is not deprecated in the following instructions:

- *LDR (immediate, ARM)* on page A8-120
- *LDR (literal)* on page A8-122
- *LDR (register)* on page A8-124
- *STR (immediate, ARM)* on page A8-384
- *STR (register)* on page A8-386.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.5 SP の汎用レジスタとしての使用

Thumb 命令セットでは、命令の制限されたセットでのみ SP(R13) を使用できます。このセットには、SP をスタックポインタとして使用する正しい方法がすべて含まれます。他の命令を、許可されたレジスタの代わりに SP でエンコードしようとすると、予測不能の動作になるか、別の命令になります。

さらに、16 ビットの CMP および ADD 命令の上位レジスタ形式で SP(R13) を Rm として使用することは推奨されません。また、SP を使用する MOV（レジスタ）の一部の形式は推奨されません。*MOV（レジスタ）* P. A8-196 を参照して下さい。

大部分の ARM 命令では、Thumb 命令の場合とは異なり、R0 ～ R12 の場合と全く同じに SP にアクセスできます。これは、SP を汎用レジスタとして使用できることを意味します。ただし、対応する Thumb 命令で SP を使用できない場合、ARM 命令で同じ方法により SP を使用することは推奨されません。

SP 操作に使用できる命令のリストを、*SP の使用が非推奨ではない ARM 命令*に示します。

D.5.1 SP の使用が非推奨ではない ARM 命令

このセクションに記載されていない ARM 命令では、SP の使用は非推奨です。

次に示す ARM データ処理命令では、一部の使用法について SP の使用は非推奨ではありません。

- *ADD（SP + イミディエート）* P. A8-28
- *ADD（SP + レジスタ）* P. A8-30
- *CMN（イミディエート）* P. A8-74
- *CMN（レジスタ）* P. A8-76
- *CMP（イミディエート）* P. A8-80
- *CMP（レジスタ）* P. A8-82
- *MOV（レジスタ）* P. A8-196
- *SUB（SP − イミディエート）* P. A8-426
- *SUB（SP − レジスタ）* P. A8-428

これらの ARM 命令で、非推奨ではない SP の使用法は、*32 ビット Thumb 命令での R13 のサポート* P. A6-4 に記載されている使用法と同じものです。

ロード / ストア / プリロード命令で、ベースレジスタとして SP を使用することは非推奨ではありません。また、次の命令で SP をデスティネーションレジスタまたはソースレジスタとして使用することは非推奨ではありません。

- *LDR（イミディエート、ARM）* P. A8-120
- *LDR（リテラル）* P. A8-122
- *LDR（レジスタ）* P. A8-124
- *STR（イミディエート、ARM）* P. A8-384
- *STR（レジスタ）* P. A8-386

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.6 Explicit use of the PC in ARM instructions

Most ARM instructions, unlike Thumb instructions, provide exactly the same access to the PC as to general-purpose registers. However, the explicit use of the PC in an ARM instruction is not usually useful, and except for specific instances that are useful, such use is deprecated.

Note

Implicit use of the PC, for example in branch instructions or load (literal) instructions, is never deprecated.

Table D-4 shows where ARM instructions can explicitly use the PC. All other explicit use of the PC is deprecated.

Table D-4 Non-deprecated uses of the PC in ARM instructions

Instruction	Non-deprecated use of PC
All load and preload instructions	As destination register or base register. [a]
ADD (immediate, ARM) on page A8-22	As destination register.
ADD (register) on page A8-24	As destination register, source register, or both.
ADD (SP plus immediate) on page A8-28	As destination register.
ADR on page A8-32	As destination register.
MOV (register) on page A8-196	As destination register or source register, but not both.[b]
SUB (immediate, ARM) on page A8-420	As destination register.
SUB (register) on page A8-422	As destination register.
SUB (SP minus immediate) on page A8-426	As destination register.
SUB (SP minus register) on page A8-428	As destination register.
SUBS PC, LR and related instructions on page B6-25	As destination register.

a. Only if the instruction description permits the register to be the PC.
b. Transfer of the PC to or update of the PC from the SP is deprecated.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.6 ARM 命令での PC の明示的な使用

大部分の ARM 命令では、Thumb 命令とは異なり、汎用レジスタと全く同様に PC にアクセスできます。ただし、ARM 命令で PC を明示的に使用することは通常役に立つことはなく、役に立つ特別な場合を除いて、明示的な使用は推奨されません。

注

例えば、分岐命令やロード（リテラル）命令において PC を暗黙的に使用することは非推奨ではありません。

ARM 命令で明示的に PC を使用できる例を、表 D-4 に示します。他の場合に PC を明示的に使用することは推奨されません。

表 D-4 ARM 命令での PC の非推奨でない使用法

命令	PC の非推奨でない使用法
すべてのロードおよびプリロード命令	デスティネーションレジスタまたはベースレジスタとして [a]
ADD（イミディエート、ARM） P. A8-22	デスティネーションレジスタとして
ADD（レジスタ） P. A8-24	デスティネーションレジスタ、ソースレジスタ、または両方として
ADD（SP + イミディエート） P. A8-28	デスティネーションレジスタとして
ADR P. A8-32	デスティネーションレジスタとして
MOV（レジスタ） P. A8-196	デスティネーションレジスタまたはソースレジスタのいずれか一方のみとして[b]
SUB（イミディエート、ARM） P. A8-420	デスティネーションレジスタとして
SUB（レジスタ） P. A8-422	デスティネーションレジスタとして
SUB（SP − イミディエート） P. A8-426	デスティネーションレジスタとして
SUB（SP − レジスタ） P. A8-428	デスティネーションレジスタとして
SUBS PC、LR、および関連する命令 P. B6-25	デスティネーションレジスタとして

a. 命令の説明で、レジスタに PC を使用することを許可している場合のみ。
b. PC の SP への転送、または PC の SP からの更新は推奨されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

D.7 Deprecated Thumb instructions

Most deprecated instructions are in the ARM instruction set. Deprecated Thumb instructions are:

- use of PC as <Rd> or <Rm> in a 16-bit ADD (SP plus register) instruction
- use of SP as <Rm> in a 16-bit ADD (SP plus register) instruction
- use of SP as <Rm> in a 16-bit CMP (register) instruction
- use of MOV (register) instructions in which both <Rd> and <Rm> are the SP or PC
- use of Rn as the lowest-numbered register in the register list of a 16-bit STM instruction with base register writeback.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

D.7 非推奨の Thumb 命令

非推奨の命令のほとんどは ARM 命令セットに含まれます。非推奨の Thumb 命令は、次のものです。

- 16 ビット ADD（SP + レジスタ）命令で、PC を〈Rd〉または〈Rm〉として使用する。
- 16 ビット ADD（SP + レジスタ）命令で、SP を〈Rm〉として使用する。
- 16 ビット CMP（レジスタ）命令で、SP を〈Rm〉として使用する。
- 〈Rd〉および〈Rm〉の両方が SP または PC である MOV（レジスタ）命令を使用する。
- 16 ビット STM 命令のレジスタリストで、Rn を最下位のレジスタとして、ベースレジスタ ライトバックとともに使用する。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Appendix E
Fast Context Switch Extension (FCSE)

This appendix describes the *Fast Context Switch Extension* (FCSE). It contains the following sections:

- *About the FCSE* on page AppxE-2
- *Modified virtual addresses* on page AppxE-3
- *Debug and trace* on page AppxE-5.

Note

- From ARMv6, use of the FCSE mechanism is deprecated. The FCSE is optional in ARMv7.
- Use of both the FCSE and the ASID based memory attribute results in UNPREDICTABLE behavior. Either the FCSE must be cleared, or all memory declared as global.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

付録 E
高速コンテキストスイッチ拡張機能 (FCSE)

この付録では、*高速コンテキストスイッチ拡張機能* (FCSE) について説明します。本章は次のセクションから構成されています。

- *FCSE について* P. 付録 E-2
- *修飾仮想アドレス* P. 付録 E-3
- *デバッグおよびトレース* P. 付録 E-5

注

- ARMv6 以降では、FCSE 機構の使用は非推奨です。ARMv7 では、FCSE はオプションです。
- FCSE および ASID ベースのメモリ属性の両方を使用すると、予測不能な動作が引き起こされます。FCSE をクリアするか、またはすべてのメモリをグローバルとして宣言する必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

E.1 About the FCSE

The *Fast Context Switch Extension* (FCSE) modifies the behavior of an ARM memory system. This modification permits multiple programs running on the ARM processor to use identical address ranges, while ensuring that the addresses they present to the rest of the memory system differ.

Normally, a swap between two software processes whose address ranges overlap requires changes to be made to the virtual-to-physical address mapping defined by the MMU translation tables, see *Translation tables* on page B3-7. It also typically causes cache and TLB contents to become invalid (because they relate to the old virtual-to-physical address mapping), and so requires caches and TLBs to be flushed. As a result, each process swap has a considerable overhead, both directly because of the cost of changing the translation tables and indirectly because of the cost of subsequently reloading caches and TLBs.

By presenting different addresses to the rest of the memory system for different software processes even when they are using identical addresses, the FCSE avoids this overhead. It also permits software processes to use identical address ranges even when the rest of the memory system does not support virtual-to-physical address mapping.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

E.1 FCSE について

高速コンテキストスイッチ拡張機能 (FCSE) は、ARM メモリシステムの動作を変更します。この変更が行われると、ARM プロセッサ上で実行されている複数のプログラムが同じアドレス範囲を使用しても、メモリシステムの他の部分に対して提示されるアドレスは異なることを保証できます。

通常、アドレス範囲がオーバラップしている 2 つのソフトウェアプロセス間におけるスワップでは、MMU 変換テーブルで定義されている仮想アドレスから物理アドレスへのマッピングを変更する必要があります。*変換テーブル* P. B3-7 を参照して下さい。また、通常はキャッシュおよび TLB の内容が（以前の仮想アドレスから物理アドレスへのマッピングに関連しているため）無効になるので、キャッシュおよび TLB をフラッシュする必要があります。結果として、プロセススワップごとに大きなオーバヘッドが発生します。これは、直接的には変換テーブルを変更するコストのため、間接的には後続のキャッシュおよび TLB のリロードのコストのためです。

FCSE は、ソフトウェアの各プロセスが同じアドレスを使用していても、メモリシステムの他の部分に対しては異なるアドレスとして提示するため、このオーバヘッドを回避できます。また、メモリシステムの他の部分で仮想アドレスから物理アドレスへのマッピングがサポートされていない場合でも、ソフトウェアプロセスが同じアドレス範囲を使用することができます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

E.2 Modified virtual addresses

The 4GB virtual address space is divided into 128 process blocks, each of size 32MB. Each process block can contain a program that has been compiled to use the address range `0x00000000` to `0x01FFFFFF`. For each of i=0 to 127, process block i runs from address (i × `0x02000000`) to address (i × `0x02000000` + `0x01FFFFFF`).

The FCSE processes each virtual address for a memory access generated by the ARM processor to produce a *modified virtual address*, that is sent to the rest of the memory system to be used in place of the normal virtual address. For an MMU-based memory system, the process is illustrated in Figure E-1:

Figure E-1 Address flow in MMU memory system with FCSE

When the ARM processor generates a memory access, the translation of the Virtual Address (VA) into the Modified Virtual Address (MVA) is described by the `FCSETranslate()` function in *FCSE translation* on page B3-156.

When the top seven bits of the address are zero, the translation replaces these bits by the value of FCSEIDR.PID when they are zero, and otherwise the translation leaves the address unchanged. When FCSEIDR.PID has its reset value of 0b0000000, the translation leaves the address unchanged, meaning that the FCSE is effectively disabled.

The value of FCSEIDR.PID is also known as the *FCSE process ID* of the current process. For more information, see *c13, FCSE Process ID Register (FCSEIDR)* on page B3-152.

The effect of setting the FCSEIDR to a nonzero value at a time when any translation table entries have enabled the alternative Context ID, ASID-based support (nG bit == 1) is UNPREDICTABLE. For more information about ASIDs see *About the VMSA* on page B3-2.

Note

Virtual addresses are sometimes passed to the memory system as data. For these operations, no address modification occurs, and MVA = VA.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

E.2 修飾仮想アドレス

4GB の仮想アドレス空間は 128 個のプロセスブロックに分割されており、それぞれのサイズは 32MB です。それぞれのプロセスブロックには、アドレス範囲 `0x00000000` 〜 `0x01FFFFFF` を使用するようにコンパイルされたプログラムが含まれています。i = 0 〜 127 のそれぞれについて、プロセスブロック i は、アドレス (i × `0x02000000`) 〜アドレス (i × `0x02000000` + `0x01FFFFFF`) で実行されます。

FCSE により、ARM プロセッサが生成したメモリアクセス用の各仮想アドレスが処理され、メモリシステムの他の部分に送信されて通常の仮想アドレスの代わりに使用される*修飾仮想アドレス*が生成されます。MMU ベースのメモリシステムについて、この処理を図 E-1 に示します。

図 E-1 FCSE を使用した MMU メモリシステムでのアドレスフロー

ARM プロセッサによりメモリアクセスが生成されるとき、仮想アドレス (VA) から修飾仮想アドレス (MVA) への変換は、*FCSE 変換* P. B3-156 の `FCSETranslate()` 関数によって記述されます。

アドレスの上位 7 ビットが 0 の場合、変換によりこれらのビットが FCSEIDR.PID の値で置き換えられ、そうでない場合は変換でアドレスは変更されません。FCSEIDR.PID がリセット値 0b0000000 である場合、変換によりアドレスは変更されません。これは、FCSE が実効上は非稼動であることを意味します。

FCSEIDR.PID の値は、現在のプロセスの *FCSE プロセス ID* とも呼ばれます。詳細については、*c13*、*FCSE プロセス ID レジスタ (FCSEIDR)* P. B3-152 を参照して下さい。

いずれかの変換テーブルエントリで、代替コンテキスト ID、ASID ベースのサポート（nG ビット == 1）が可能になっているときに、FCSEIDR を 0 以外の値に設定すると、影響は予測不能です。ASID の詳細については、*VMSA* についてP. B3-2 を参照して下さい。

注

仮想アドレスは、メモリシステムにデータとして渡される場合があります。これらの操作では、アドレスは変更されず MVA = VA です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Each process is compiled to use the address range `0x00000000` to `0x01FFFFFF`. When referring to its own instructions and data, therefore, the program generates VAs whose top seven bits are all zero. The resulting MVAs have their top seven bits replaced by FCSEIDR.PID, and so lie in the process block of the current process.

The program can also generate VAs whose top seven bits are not all zero. When this happens, the MVA is equal to the VA. This enables the program to address the process block of another process, provided the other process does not have process ID 0. Provided access permissions are set correctly, this can be used for inter-process communication.

Note

ARM recommends that only process IDs 1 and above are used for general-purpose processes, because the process with process ID 0 cannot be communicated with in this fashion.

Use of the FCSE therefore reduces the cost of a process swap to:

- The cost of a write of the FCSEIDR.PID.
- The cost of changing access permissions if they need changing for the new process. In an MMU-based system, this might involve changing the translation table entries individually, or pointing to a new translation table by changing one or more of TTBR0, TTBR1, and TTBCR. Any change to the translation tables is likely to involve invalidation of the TLB entries affected. However, this is usually significantly cheaper than the cache flush that would be required without the FCSE. Also, in some cases, changes to the translation table, and the associated explicit TLB management, can be avoided by the use of domains. This reduces the cost to that of a write to the Domain Access Control Register, see *Domains* on page B3-31.

The FCSE is deprecated. The use of cache, branch predictor and TLB operations with MVA based addresses that, as a result of the Multiprocessing Extensions, would affect other processors as described in section 3.2 is UNPREDICTABLE if FCSEIDR.PID is not zero.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

それぞれのプロセスは、アドレス範囲 `0x00000000` 〜 `0x01FFFFFF` を使用するようにコンパイルされます。このため、独自の命令およびデータを参照する場合、プログラムは上位 7 ビットがすべて 0 である VA を生成します。結果として生成される MVA では、上位 7 ビットが FCSEIDR.PID で置き換えられ、現在のプロセスのプロセスブロックを指すようになります。

プログラムは、上位 7 ビットのすべてが 0 ではない VA を生成することもできます。この場合、MVA は VA に等しくなります。これにより、他のプロセスのプロセス ID が 0 ではない場合、プログラムが別のプロセスのプロセスブロックをアドレス指定することが可能になります。アクセス許可が正しく設定されている場合は、プロセス間の通信に使用することができます。

注

汎用プロセスのプロセス ID には 1 以上のみを使用することをお勧めします。プロセス ID が 0 のプロセスでは、この方法で通信することができないためです。

このため、FCSE を使用すると、プロセススワップにおいて次のようなコストが低減されます。

- FCSEIDR.PID の書き込みのコスト
- 新規プロセスについてアクセス許可の変更が必要な場合、その変更のコスト。MMU ベースのシステムでは、これに、変換テーブルエントリの個別の変更、または TTBR0、TTBR1、および TTBCR の 1 つ以上を変更することによる新規変換テーブルの指定が含まれる場合があります。変換テーブルの変更には、影響を受ける TLB エントリの無効化が含まれることがあります。ただし、これは通常、FCSE がない場合に必要になるキャッシュフラッシュよりもかなりコストが低くなります。また、一部の場合においては、変換テーブルの変更、および関連する明示的な TLB 管理は、ドメインを使用して回避できます。これにより、ドメインアクセス制御レジスタへの書き込みのコストが低減されます。*ドメイン* P. B3-31 を参照して下さい。

FCSE は推奨されません。セクション 3.2 で説明されているように、マルチプロセッシング拡張機能の結果として、他のプロセッサに影響を与える可能性がある MVA ベースのアドレスを使用するキャッシュ、分岐予測器、TLB の操作を使用した結果は、FCSEIDR.PID が 0 以外の場合予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

E.3 Debug and trace

It is IMPLEMENTATION DEFINED whether a VA or MVA is used by breakpoint and watchpoint mechanisms. However, ARM strongly recommends that any implementation that includes the FCSE uses MVAs, to avoid trigger aliasing.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

E.3 デバッグおよびトレース

ブレークポイントおよびウォッチポイント機構で VA と MVA のどちらが使用されるかは実装定義です。ただし、FCSE を含む実装では、エイリアシングのトリガを回避するために MVA の使用を強くお勧めします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Appendix F
VFP Vector Operation Support

This appendix provides reference information about VFP vector operation.

This appendix contains the following sections:

- *About VFP vector mode* on page AppxF-2
- *Vector length and stride control* on page AppxF-3
- *VFP register banks* on page AppxF-5
- *VFP instruction type selection* on page AppxF-7.

Note

The use of VFP vector mode is deprecated. This information is provided for backwards compatibility only.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

付録 F
VFP ベクタ演算のサポート

この付録では、VFP ベクタ演算に関する参照情報について説明します。

この付録は次のセクションから構成されています。

- *VFP ベクタモードについて* P. 付録 F-2
- *ベクタ長とストライドの制御* P. 付録 F-3
- *VFP レジスタバンク* P. 付録 F-5
- *VFP 命令タイプの選択* P. 付録 F-7

注

VFP ベクタモードの使用は推奨されません。この情報は、下位互換性の目的でのみ提供されるものです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

F.1 About VFP vector mode

The single-precision registers can be used to hold short vectors of up to 8 single-precision values. Arithmetic operations on all the elements of such a vector can be specified by just one single-precision arithmetic instruction.

Similarly, the double-precision registers can be used to hold short vectors of up to 4 double-precision values, and double-precision arithmetic instructions can specify operations on these vectors.

A vector consists of 2-8 registers from a single *bank*. *VFP register banks* on page AppxF-5 describes the division of the VFP register set into banks.

The FPSCR.LEN field controls the number of elements in a vector. The register number in the instruction specifies the register that contains the first element of the vector. The FPSCR.STRIDE field controls the increment between the register numbers of the elements of the vector. If the total increment causes the register number to overflow the top of a register bank, the register number wraps around to the bottom of the bank, as shown in *VFP register banks* on page AppxF-5.

For details of the FPSCR.LEN and FPSCR.STRIDE fields see *Vector length and stride control* on page AppxF-3.

A VFP instruction can operate on:

- operand vectors with LEN elements, producing a result vector with LEN elements
- an operand vector with LEN elements and a scalar operand, producing a result vector with LEN elements
- scalar operands, producing a scalar result.

These three operation types are identical if LEN == 1.

To control which type of operation an instruction performs, you choose the registers for the instruction from different register banks. *VFP instruction type selection* on page AppxF-7 describes how to select the instruction type.

F.1.1 Affected instructions

The following VFP instructions are affected by VFP vector mode:

VABS	VADD	VDIV	VMLA	VMLS
VMOV (immediate)	VMOV (register)	VMUL	VNEG	VNMLA
VNMLS	VNMUL	VSQRT	VSUB	

All other VFP instructions behave as described in their instruction descriptions regardless of the values of FPSCR.LEN and FPSCR.STRIDE.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

F.1 VFP ベクタモードについて

単精度レジスタは、最大 8 つの単精度値のショートベクタを保持するために使用できます。このようなベクタのすべてのエレメントに関する算術演算は、ただ 1 つの単精度算術命令によって指定できます。

同様に、倍精度レジスタは、最大 4 つの倍精度値のショートベクタを保持するために使用でき、倍精度算術命令では、これらのベクタに関する演算を指定できます。

1 つのベクタは、単一の*バンク*に含まれる 2 つ～ 8 つのレジスタで構成されます。VFP レジスタセットのバンクへの分割については、*VFP レジスタバンク* P. 付録 F-5 を参照して下さい。

FPSCR.LEN フィールドは、1 つのベクタに含まれるエレメントの数を制御します。命令のレジスタ番号により、ベクタの最初のエレメントが含まれるレジスタが指定されます。
FPSCR.STRIDE フィールドにより、ベクタのエレメントのレジスタ番号間の増分が制御されます。増分の合計によりレジスタ番号がレジスタバンクの最上位をオーバフローした場合は、*VFP レジスタバンク* P. 付録 F-5 に示されるように、レジスタ番号はバンクの最下位にラップアラウンドされます。

FPSCR.LEN および FPSCR.STRIDE フィールドの詳細については、*ベクタ長とストライドの制御* P. 付録 F-3 を参照して下さい。

VFP 命令は、次のオペランドで動作することができます。

- LEN エレメントを持つオペランドベクタ。LEN エレメントを持つ結果ベクタを生成します。
- LEN エレメントを持つオペランドベクタとスカラオペランド。LEN エレメントを持つ結果ベクタを生成します。
- スカラオペランド。スカラ結果を生成します。

LEN == 1 の場合、これら 3 つの演算タイプは同一です。

命令が実行する演算のタイプを制御するには、別のレジスタバンクから命令用のレジスタを選択します。命令タイプの選択方法については、*VFP 命令タイプの選択* P. 付録 F-7 で説明されています。

F.1.1 影響を受ける命令

VFP ベクタモードでは、次の VFP 命令が影響を受けます。

VABS	VADD	VDIV	VMLA	VMLS
VMOV（イミディエート）	VMOV（レジスタ）	VMUL	VNEG	VNMLA
VNMLS	VNMUL	VSQRT	VSUB	

他のすべての VFP 命令は、FPSCR.LEN および FPSCR.STRIDE の値にかかわらず、それぞれの命令の説明に記述されているように動作します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

F.2 Vector length and stride control

The FPSCR.LEN field, bits [18:16], controls the vector length for VFP instructions that operate on short vectors, that is, how many registers are in a vector operand. Similarly, the FPSCR.STRIDE field, bits [21:20], controls the vector stride, that is, how far apart the registers in a vector lie in the register bank. For information about the FPSCR see *Floating-point Status and Control Register (FPSCR)* on page A2-28.

The permitted combinations of LEN and STRIDE are shown in Table F-1. All other combinations of LEN and STRIDE produce UNPREDICTABLE results.

The combination LEN == 0b000, STRIDE == 0b00 is called *scalar mode*. When it is in effect, all arithmetic instructions specify scalar operations. Otherwise, most arithmetic instructions specify a scalar operation if their destination is in the range:

- S0-S7 for a single-precision operation
- D0-D3 or D16-D19 for a double-precision operation.

The full rules used to determine which operands are vectors and full details of how vector operands are specified can be found in *VFP instruction type selection* on page AppxF-7.

The rules for vector operands do not permit the same register to appear twice or more in a vector. The permitted LEN and STRIDE combinations listed in Table F-1 never cause this to happen for single-precision instructions, so single-precision scalar and vector instructions can be used with all of these LEN and STRIDE combinations.

For double-precision vector instructions, some of the permitted LEN and STRIDE combinations would cause the same register to appear twice in a vector. If a double-precision vector instruction is executed with such a LEN and STRIDE combination in effect, the instruction is UNPREDICTABLE. The last column of Table 2-2 indicates which LEN and STRIDE combinations this applies to. Double-precision scalar instructions work normally with all of the permitted LEN and STRIDE combinations.

Table F-1 Vector length and stride combinations

LEN	STRIDE	Vector length	Vector stride	Double-precision vector instructions
0b000	0b00	1	-	All instructions are scalar
0b001	0b00	2	1	Work as described in this appendix
0b001	0b11	2	2	Work as described in this appendix
0b010	0b00	3	1	Work as described in this appendix
0b010	0b11	3	2	UNPREDICTABLE
0b011	0b00	4	1	Work as described in this appendix
0b011	0b11	4	2	UNPREDICTABLE
0b100	0b00	5	1	UNPREDICTABLE

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

F.2 ベクタ長とストライドの制御

FPSCR.LEN フィールド（ビット [18:16]）は、ショートベクタで動作する VFP 命令のベクタ長、つまりベクタオペランドに含まれるレジスタの数を制御します。同様に、FPSCR.STRIDE フィールド（ビット [21:20]）は、ベクタストライド、つまりベクタに含まれるレジスタのレジスタバンク内での距離を制御します。FPSCR の詳細については、*浮動小数点ステータスおよび制御レジスタ (FPSCR)* P. A2-28 を参照して下さい。

LEN と STRIDE の許容される組み合わせを、表 F-1 に示します。LEN と STRIDE の他のすべての組み合わせでは、予測不能の結果が引き起こされます。

LEN == 0b000、STRIDE == 0b00 の組み合わせは、*スカラモード*と呼ばれます。この場合、すべての算術命令でスカラ演算が指定されます。それ以外の場合、ほとんどの算術命令について、デスティネーションが次の範囲内の場合にスカラ演算が指定されます。

- S0 ～ S7（単精度演算の場合）
- D0 ～ D3 または D16 ～ D19（倍精度演算の場合）

どのオペランドがベクタであるかを決定するために使用される完全な規則、およびベクタオペランドの指定方法の詳細については、*VFP 命令タイプの選択* P. 付録 F-7 を参照して下さい。

ベクタオペランドの規則では、1 つのベクタに同じレジスタが 2 回以上出現することは許可されていません。表 F-1 に示されている LEN と STRIDE の許容される組み合わせでは、単精度命令でこれが発生することはありません。このため、単精度のスカラおよびベクタ命令を、LEN と STRIDE のすべての組み合わせで使用することができます。

倍精度のベクタ命令の場合、LEN と STRIDE の許容される組み合わせの一部では、1 つのベクタに同じレジスタが 2 回出現する場合があります。倍精度のベクタ命令がこのような LEN と STRIDE の組み合わせで実行された場合、命令は予測不能です。表 2-2 の最後の列では、これが適用される LEN と STRIDE の組み合わせが示されています。倍精度のスカラ命令は、LEN と STRIDE の許容されるすべての組み合わせで正常に動作します。

表 F-1 ベクタ長とストライドの組み合わせ

LEN	STRIDE	ベクタ長	ベクタストライド	倍精度ベクタ命令
0b000	0b00	1	–	すべての命令がスカラ
0b001	0b00	2	1	この付録で記述されているように動作
0b001	0b11	2	2	この付録で記述されているように動作
0b010	0b00	3	1	この付録で記述されているように動作
0b010	0b11	3	2	予測不能
0b011	0b00	4	1	この付録で記述されているように動作
0b011	0b11	4	2	予測不能
0b100	0b00	5	1	予測不能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table F-1 Vector length and stride combinations (continued)

LEN	STRIDE	Vector length	Vector stride	Double-precision vector instructions
0b101	0b00	6	1	UNPREDICTABLE
0b110	0b00	7	1	UNPREDICTABLE
0b111	0b00	8	1	UNPREDICTABLE

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 F-1 ベクタ長とストライドの組み合わせ （続き）

LEN	STRIDE	ベクタ長	ベクタストライド	倍精度ベクタ命令
0b101	0b00	6	1	予測不能
0b110	0b00	7	1	予測不能
0b111	0b00	8	1	予測不能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

F.3 VFP register banks

The Advanced SIMD and VFP registers are divided into banks as follows:

- The single-precision registers are divided into four banks of eight. This is shown in Figure F-1. The first bank is a *scalar* bank, and the other three are *vector* banks.
- In a processor with 32 double-precision registers, the double-precision registers are divided into eight banks of four. This is shown in Figure F-2. The first and fifth banks are scalar banks, and the other six are vector banks.
- In a processor with 16 double-precision registers, the double-precision registers are divided into four banks of four. This is shown in Figure F-3 on page AppxF-6. The first bank is a scalar bank, and the other three are vector banks.

Figure F-1 Single-precision register banks

Figure F-2 Register banks, 32 double-precision register VFP

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

F.3 VFP レジスタバンク

アドバンスト SIMD および VFP のレジスタは、次のようにバンクに分割されています。

- 単精度レジスタは、8 つずつ 4 つのバンクに分割されています。これを、図 F-1 に示します。最初のバンクはスカラバンクで、他の 3 つのバンクはベクタバンクです。
- 32 個の倍精度レジスタを持つプロセッサでは、倍精度レジスタは 4 つずつ 8 つのバンクに分割されています。これを、図 F-2 に示します。最初と 5 番目のバンクはスカラバンクで、他の 6 つのバンクはベクタバンクです。
- 16 個の倍精度レジスタを持つプロセッサでは、倍精度レジスタは 4 つずつ 4 つのバンクに分割されています。これを、P. 付録 F-6 の図 F-3 に示します。最初のバンクはスカラバンクで、他の 3 つのバンクはベクタバンクです。

図 F-1 単精度レジスタバンク

図 F-2 レジスタバンク、32 個の倍精度レジスタ VFP

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Figure F-3 Register banks, 16 double-precision register VFP

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

図 F-3 レジスタバンク、16 個の倍精度レジスタ VFP

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

F.4 VFP instruction type selection

Table F-2 shows how the selection of registers in an instruction controls the operation of the instruction.

Table F-2

Destination register bank	1st operand bank	2nd operand bank	Destination type	1st operand type	2nd operand type
Scalar	Any	Any	Scalar	Scalar	Scalar
Vector	Any	Scalar	Vector	Vector	Scalar
Vector	Any	Vector	Vector	Vector	Vector
Scalar	Any	None	Scalar	Scalar	-
Vector	Scalar	None	Vector	Scalar	-
Vector	Vector	None	Vector	Vector	-

- If the instruction has two operands:
  - — If the destination register is in a scalar register bank, the operands and result are all scalars.
  - — If the destination register is in a vector register bank and the second operand is in a scalar bank, the second operand is a scalar, but both the destination and the first operand are vectors. Each element of the result is produced by an operation on the corresponding element of the first operand and the same scalar.
  - — If the destination register and the second operand are both in vector register banks, the operands and result are all vectors. Each element of the result is produced by an operation on corresponding elements of both operands.
- If the instruction has one operand:
  - — If the destination register is in a scalar register bank, the operand and result are both scalars.
  - — If the destination register is in a vector register bank and the operand is in a scalar bank, the result is a vector and the operand is a scalar. The result is duplicated to each element of the destination vector.
  - — If the destination register and the operand are both in vector register banks, the operand and result are both vectors. Each element of the result is produced by an operation on the corresponding element of the operand.

Some VFP instructions have three operands, but in these cases one of the operand vectors is also the result vector. They operate in the same way as two operand instructions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

F.4 VFP 命令タイプの選択

命令内のレジスタを選択することにより命令の動作を制御する方法を、表 F-2 に示します。

表 F-2

デスティネーションレジスタバンク	最初のオペランドバンク	2 番目のオペランドバンク	デスティネーションタイプ	最初のオペランドタイプ	2 番目のオペランドタイプ
スカラ	任意	任意	スカラ	スカラ	スカラ
ベクタ	任意	スカラ	ベクタ	ベクタ	スカラ
ベクタ	任意	ベクタ	ベクタ	ベクタ	ベクタ
スカラ	任意	なし	スカラ	スカラ	–
ベクタ	スカラ	なし	ベクタ	スカラ	–
ベクタ	ベクタ	なし	ベクタ	ベクタ	–

- 命令が 2 つのオペランドを持つ場合、次の規則が適用されます。
  - – デスティネーションレジスタがスカラ レジスタバンクにある場合、オペランドおよび結果はすべてスカラです。
  - – デスティネーションレジスタがベクタ レジスタバンクにあり、2 番目のオペランドがスカラバンクにある場合、2 番目のオペランドはスカラですが、デスティネーションおよび最初のオペランドは両方ともベクタです。結果の各エレメントは、最初のオペランドの対応するエレメントと、同一のスカラとの演算により生成されます。
  - – デスティネーションレジスタおよび 2 番目のオペランドが両方ともベクタ レジスタバンクにある場合、オペランドおよび結果はすべてベクタです。結果の各エレメントは、両方のオペランドの対応するエレメントへの演算により生成されます。
- 命令が 1 つのオペランドを持つ場合、次の規則が適用されます。
  - – デスティネーションレジスタがスカラ レジスタバンクにある場合、オペランドおよび結果は両方ともスカラです。
  - – デスティネーションレジスタがベクタ レジスタバンクにあり、オペランドがスカラバンクにある場合、結果はベクタでオペランドはスカラです。結果は、デスティネーションベクタの各エレメントに複製されます。
  - – デスティネーションレジスタおよびオペランドが両方ともベクタ レジスタバンクにある場合、オペランドおよび結果は両方ともベクタです。結果の各エレメントは、オペランドの対応するエレメントへの演算によって生成されます。

一部の VFP 命令には 3 つのオペランドがありますが、この場合オペランドベクタのいずれかは結果ベクタでもあります。これらの命令は、2 オペランドの命令の場合と同様に動作します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Appendix G
ARMv6 Differences

This appendix describes how ARMv6 differs from ARMv7. The appendix contains the following sections:

- *Introduction to ARMv6* on page AppxG-2
- *Application level register support* on page AppxG-3
- *Application level memory support* on page AppxG-6
- *Instruction set support* on page AppxG-10
- *System level register support* on page AppxG-16
- *System level memory model* on page AppxG-20
- *System Control coprocessor (CP15) support* on page AppxG-29.

Note

In this appendix, the description ARMvN refers to all architecture variants of ARM architecture vN described in this manual. In particular, ARMv6 refers to ARMv6, ARMv6K, and ARMv6T2, including ARMv6K with the Security Extensions. Where the description ARMvN also describes a specific architecture variant, this variant is sometimes described as the *base architecture*, for example the ARMv6 base architecture.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

付録 G
ARMv6 との相違点

この付録では、ARMv6 と ARMv7 との相違点について説明します。この付録は次のセクションから構成されています。

- *ARMv6 の概要* P. 付録 G-2
- *アプリケーションレベルのレジスタサポート* P. 付録 G-3
- *アプリケーションレベルのメモリサポート* P. 付録 G-6
- *命令セットのサポート* P. 付録 G-10
- *システムレベルのレジスタサポート* P. 付録 G-16
- *システムレベルのメモリモデル* P. 付録 G-20
- *システム制御コプロセッサ (CP15) のサポート* P. 付録 G-29

注

この付録で、ARMvN という記述は、本書で説明されている ARM アーキテクチャ vN のすべてのアーキテクチャバリアントを意味します。特に、ARMv6 という記述は、セキュリティ拡張機能を持つ ARMv6K を含む、ARMv6、ARMv6K、および ARMv6T2 を意味します。ARMvN という記述が特定のアーキテクチャバリアントを示す場合、このバリアントは*基本アーキテクチャ*、例えば ARMv6 基本アーキテクチャと記述されることがあります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.1 Introduction to ARMv6

This appendix describes the differences in the ARMv6 architecture, compared to the description of ARMv7 given in parts A and B of this manual. Key changes introduced in ARMv7 are:

- Introduction of hierarchical cache support.
- Formalizing the alternative memory system architectures into different architecture profiles:
  - the *Virtual Memory System Architecture* (VMSA) is formalized into the ARMv7-A profile
  - the *Protected Memory System Architecture* (PMSA) is formalized into the ARMv7-R profile.
- Introduction of the Advanced SIMD extensions.
- Introduction of the Thumb Execution Environment (ThumbEE). ThumbEE is required in ARMv7-A, and optional in ARMv7-R.

This appendix summarizes the features supported in ARMv6, highlighting:

- the similarities and differences with respect to ARMv7, including the following architecture variants and extensions:
  - the Security Extensions
  - the extension of the Thumb instruction set using Thumb-2 technology, introduced in ARMv6T2
  - the enhanced kernel support introduced in ARMv6K.
- legacy support for ARMv4 and ARMv5.

G.1.1 Debug

Part C of this manual describes ARMv6 Debug, ARMv7 Debug, and the differences between them.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.1 ARMv6 の概要

この付録では、本書のパート A およびパート B に記載されている ARMv7 に関する説明と比較して、ARMv6 アーキテクチャでの相違点について説明します。ARMv7 で導入された主な変更点は次のとおりです。

- 階層キャッシュサポートの導入
- 各メモリシステム アーキテクチャが、異なるアーキテクチャプロファイルとして制式化されました。
  - *仮想メモリシステム アーキテクチャ* (VMSA) は、ARMv7-A プロファイルとして制式化されました。
  - *保護メモリシステム アーキテクチャ* (PMSA) は、ARMv7-R プロファイルとして制式化されました。
- アドバンスト SIMD 拡張機能の導入
- Thumb 実行環境 (ThumbEE) の導入。ThumbEE は ARMv7-A では必須ですが、ARMv7-R ではオプションです。

この付録では、ARMv6 でサポートされる機能を要約しています。特に、次のものに重点が置かれています。

- 次のようなアーキテクチャバリアントおよび拡張機能を含む、ARMv7 と比較した場合の類似点および相違点
  - セキュリティ拡張機能
  - ARMv6T2 で導入された Thumb-2 テクノロジを使用した Thumb 命令セットの拡張
  - ARMv6K で導入された拡張カーネルサポート
- 過去の ARMv4 および ARMv5 のサポート

G.1.1 デバッグ

本書のパート C では、ARMv6 デバッグ、ARMv7 デバッグ、およびそれらの間の相違点について説明されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.2 Application level register support

The ARMv6 core registers are the same as the ARMv7 core registers. For more information, see *ARM core registers* on page A2-11. The following sections give more information about ARMv6 application level register support:

- *APSR support*
- *Instruction set state*.

G.2.1 APSR support

Application Program Status Register (APSR) support in ARMv6 is identical to ARMv7. Program status is reported in the 32-bit APSR. The format of the APSR is:

31	30	29	28	27	26 – 24	23 – 20	19 – 16	15 – 0
N	Z	C	V	Q	RAZ/SBZP	Reserved	GE[3:0]	Reserved

See *The Application Program Status Register (APSR)* on page A2-14 for the APSR bit definitions.

Earlier versions of this manual do not use the term APSR. They refer to the APSR as the CPSR with restrictions on reserved fields determined by whether the access to the register was privileged or not.

G.2.2 Instruction set state

Instruction set state support in ARMv6 is in general the same as the support available in ARMv7. The only differences are that:

- ThumbEE state is not supported in ARMv6. It is introduced in ARMv7.
- In ARMv6 and ARMv6K, but not in ARMv6T2, when the processor is in a privileged mode you must take care not to attempt to change the instruction set state by writing nonzero values to CPSR.J and CPSR.T with an `MSR` instruction. For more information, see *Format of the CPSR and SPSRs* on page AppxG-17.

All ARMv6 implementations support the ARM instruction set. The ARMv6 base architecture and ARMv6K also support a subset of the Thumb instruction set that can be executed entirely as 16-bit instructions. The only 32-bit instructions in this subset are restricted-range versions of the `BL` and `BLX` (immediate) instructions. See *BL and BLX (immediate) instructions, before ARMv6T2* on page AppxG-4 for a description of how these instructions can be executed as 16-bit instructions.

The supported ARM and Thumb instructions in the ARMv6 base architecture and ARMv6K are summarized in *Instruction set support* on page AppxG-10, and the instruction descriptions in Chapter A8 *Instruction Details* give details of the architecture variants that support each instruction encoding.

Jazelle state is supported as in ARMv7. For more information, see:

- *Jazelle direct bytecode execution support* on page A2-73, for application level information
- *Jazelle direct bytecode execution* on page B1-74, for system level information.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.2 アプリケーションレベルのレジスタサポート

ARMv6 のコアレジスタは、ARMv7 のコアレジスタと同じです。詳細については、*ARM コアレジスタ* P. A2-11 を参照して下さい。次に示すセクションでは、ARMv6 におけるアプリケーションレベルのレジスタサポートの詳細について説明します。

- *APSR のサポート*
- *命令セット状態*

G.2.1 APSR のサポート

ARMv6 での *アプリケーションプログラム ステータスレジスタ* (APSR) のサポートは、ARMv7 の場合と同じです。プログラムステータスは、32 ビットの APSR で報告されます。APSR のフォーマットは次のとおりです。

31	30	29	28	27	26 – 24	23 – 20	19 – 16	15 – 0
N	Z	C	V	Q	RAZ/SBZP	予約	GE[3:0]	予約

APSR のビットの定義については、*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。

本書の初期のバージョンでは、APSR という用語は使用されていません。初期のバージョンでは、レジスタへのアクセスが特権付きであるかどうかによって決定される、予約フィールドに関する制限があるため、APSR は CPSR と呼ばれています。

G.2.2 命令セット状態

ARMv6 での命令セット状態のサポートは、一般的には ARMv7 で使用可能なサポートと同じです。相違点は次のもののみです。

- ThumbEE 状態は、ARMv6 ではサポートされません。この状態は ARMv7 で導入されています。
- ARMv6 および ARMv6K（ARMv6T2 は除く）では、プロセッサが特権モードの場合、`MSR` 命令により CPSR.J および CPSR.T に 0 以外の値を書き込むことで命令セット状態を変更する操作を行わないように注意する必要があります。詳細については、*CPSR と SPSR のフォーマット* P. 付録 G-17 を参照して下さい。

すべての ARMv6 実装で、ARM 命令セットがサポートされます。ARMv6 基本アーキテクチャおよび ARMv6K では、完全に 16 ビット命令として実行できる Thumb 命令セットのサブセットもサポートされます。このサブセットに含まれる 32 ビット命令は、`BL` および `BLX`（イミディエート）命令の範囲制限付きのバージョンのみです。これらの命令を 16 ビット命令として実行する方法の説明については、*ARM v 6T2 以前の BL および BLX（イミディエート）命令* P. 付録 G-4 を参照して下さい。

ARMv6 基本アーキテクチャおよび ARMv6K でサポートされる ARM 命令および Thumb 命令の要約を、*命令セットのサポート* P. 付録 G-10 に示します。第 A8 章*「命令の詳細」*の命令に関する説明では、各命令エンコードをサポートするアーキテクチャバリアントの詳細が示されています。

Jazelle 状態は、ARMv7 と同様にサポートされます。詳細については、次のセクションを参照して下さい。

- *Jazelle 直接バイトコード実行のサポート* P. A2-73（アプリケーションレベルの情報）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARMv6T2 supports the full Thumb instruction set, apart from a few instructions that are introduced in ARMv7.

Interworking

In ARMv6, the instructions that provide interworking branches between ARM and Thumb states are:

- BL and BLX
- LDR, LDM, and POP instructions that load the PC.

In ARMv7, the following ARM instructions also perform interworking branches if their destination register is the PC and the 'S' option is not specified:

- ADC, ADD, AND, ASR, BIC, EOR, LSL, LSR, MOV, MVN, ORR, ROR, RRX, RSB, RSC, SBC, and SUB.

The instructions do not perform interworking branches in ARMv6, and the corresponding Thumb instructions do not perform interworking branches in either ARMv6 or ARMv7. This functionality is described by the ALUWritePC() pseudocode function. See *Pseudocode details of operations on ARM core registers* on page A2-12.

BL and BLX (immediate) instructions, before ARMv6T2

In ARMv4T, ARMv5T, ARMv5TE, ARMv5TEJ, ARMv6, and ARMv6K, the BL and BLX (immediate) instructions are the only 32-bit Thumb instructions, and the maximum range of the branches that they specify is restricted to approximately +/-4MB. This means that each of the two halfwords of these instructions has top five bits 0b11101, 0b11110, or 0b11111, and makes it possible to execute the two halfwords as separate 16-bit instructions.

The following descriptions use the format described in *Instruction encodings* on page A8-2, except that they:

- name the encodings H1, H2 and H3
- have pseudocode that defines the entire operation of the instruction, instead of separate encoding-specific pseudocode and Operation pseudocode.

When the two halfwords of a BL or BLX (immediate) instruction are executed separately, their behavior is as follows:

Encoding H1 ARMv4T, ARMv5T*, ARMv6, ARMv6K Used for BL and BLX
BL{X} <label> First of two 16-bit instructions

15 14 13 12 11	10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0	imm11

```
LR = PC + SignExtend(imm11:Zeros(12), 32);
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- *Jazelle 直接バイトコード実行* P. B1-74（システムレベルの情報）

ARMv6T2 では、ARMv7 で導入されたいくつかの命令を除いて、完全な Thumb 命令セットがサポートされます。

インターワーキング

ARMv6 では、ARM 状態と Thumb 状態との間のインターワーキング分岐を提供する命令は、次のものです。

- BL および BLX
- PC をロードする LDR、LDM、POP 命令

ARMv7 では、デスティネーションレジスタが PC で、「S」オプションが指定されていない場合、次の ARM 命令でもインターワーキング分岐が実行されます。

- ADC、ADD、AND、ASR、BIC、EOR、LSL、LSR、MOV、MVN、ORR、ROR、RRX、RSB、RSC、SBC、SUB

ARMv6 ではこれらの命令によりインターワーキング分岐は実行されません。ARMv6 および ARMv7 のどちらでも、対応する Thumb 命令ではインターワーキング分岐は実行されません。この機能は、ALUWritePC() 擬似コード関数で記述されます。詳細については、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 を参照して下さい。

ARM v 6T2 以前の BL および BLX（イミディエート）命令

ARMv4T、ARMv5T、ARMv5TE、ARMv5TEJ、ARMv6、ARMv6K では、BL および BLX（イミディエート）命令のみが 32 ビットの Thumb 命令で、その命令で指定される分岐の最大範囲は約 +/-4MB に制限されます。これは、命令の 2 つのハーフワードそれぞれの上位 5 ビットが 0b11101、0b11110、または 0b11111 であり、2 つのハーフワードを別の 16 ビット命令として実行可能であることを意味します。

以下の説明では、次の点を除いて、*命令のエンコード* P. A8-2 で説明されているフォーマットが使用されます。

- エンコードは H1、H2、H3 という名前で呼ばれます。
- 個別のエンコード固有の擬似コードおよび動作擬似コードではなく、命令全体の動作を定義する擬似コードが記載されています。

BL または BLX（イミディエート）命令の 2 つのハーフワードを個別に実行する場合、その動作は次のとおりです。

エンコード H1 ARMv4T、ARMv5T*、ARMv6、ARMv6K BL および BLX で使用される
BL{X} <label> 2 つの 16 ビット命令のうち最初の命令

15 14 13 12 11	10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 0	imm11

```
LR = PC + SignExtend(imm11:Zeros(12), 32);
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Encoding H2 ARMv4T, ARMv5T*, ARMv6, ARMv6K Used for BL

BL <label> Second of two 16-bit instructions

15	14	13	12	11	10 9 8 7 6 5 4 3 2 1 0
1	1	1	1	1	imm11

```
next_instr_addr = PC - 2;
BranchWritePC(LR + ZeroExtend(imm11:'0', 32));
LR = next_instr_addr<31:1> : '1';
```

Encoding H3 ARMv5T*, ARMv6, ARMv6K Used for BLX

BLX <label> Second of two 16-bit instructions

15	14	13	12	11	10 9 8 7 6 5 4 3 2 1	0
1	1	1	0	1	imm10	op

```
if op == '0' then
    next_instr_addr = PC - 2;
    SelectInstrSet(InstrSet_ARM);
    BranchWritePC(Align(LR,4) + ZeroExtend(imm10:'00', 32));
    LR = next_instr_addr;
else
    UNDEFINED;
```

An encoding H1 instruction must be followed by an encoding H2 or encoding H3 instruction. Similarly, an encoding H2 or encoding H3 instruction must be preceded by an encoding H1 instruction. Otherwise, the behavior is UNPREDICTABLE.

It is IMPLEMENTATION DEFINED whether processor exceptions can occur between the two instructions of a BL or BLX pair. If they can, the ARM exception return instructions must be able to return correctly to the second instruction of the pair. The exception handler does not have to take special precautions. See *Exception return* on page B1-38 for the definition of exception return instructions.

Note

There are no Thumb exception return instructions in the architecture versions that support separate execution of the two halfwords of BL and BLX (immediate) instructions. Also, the ARM RFE instruction is only defined from ARMv6 onwards.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

エンコード H2 ARMv4T、ARMv5T*、ARMv6、ARMv6K BL で使用される

BL <label> 2 つの 16 ビット命令のうち 2 番目の命令

15	14	13	12	11	10 9 8 7 6 5 4 3 2 1 0
1	1	1	1	1	imm11

```
next_instr_addr = PC - 2;
BranchWritePC(LR + ZeroExtend(imm11:'0', 32));
LR = next_instr_addr<31:1> : '1';
```

エンコード H3 ARMv5T*、ARMv6、ARMv6K BLX で使用される

BLX <label> 2 つの 16 ビット命令のうち 2 番目の命令

15	14	13	12	11	10 9 8 7 6 5 4 3 2 1	0
1	1	1	0	1	imm10	op

```
if op == '0' then
    next_instr_addr = PC - 2;
    SelectInstrSet(InstrSet_ARM);
    BranchWritePC(Align(LR,4) + ZeroExtend(imm10:'00', 32));
    LR = next_instr_addr;
else
    UNDEFINED;
```

エンコード H2 またはエンコード H3 の命令は、エンコード H1 の命令の後に配置する必要があります。同様に、エンコード H1 の命令は、エンコード H2 またはエンコード H3 の命令の前に配置される必要があります。そうでない場合、動作は予測不能です。

BL または BLX の 2 つの命令ペア間でプロセッサ例外が発生する場合があるかどうかは実装定義です。発生する場合、ARM の例外からの復帰命令は、ペアの 2 番目の命令に正確に戻ることができる必要があります。例外ハンドラでは、特別な注意は必要はありません。例外からの復帰命令の定義については、*例外からの復帰* P. B1-38 を参照して下さい。

注

BL および BLX（イミディエート）命令の 2 つのハーフワードが個別に実行されることをサポートするアーキテクチャバージョンには、Thumb の例外からの復帰命令はありません。また、ARM RFE 命令は、ARMv6 および以降でのみ定義されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.3 Application level memory support

Memory support covers address alignment, endian support, semaphore support, memory order model, caches, and write buffers. The following sections give an application level description of ARMv6 memory support:

- *Alignment*
- *Endian support* on page AppxG-7
- *Semaphore support* on page AppxG-8
- *Memory model and memory ordering* on page AppxG-8.

G.3.1 Alignment

ARMv6 supports:

- a legacy alignment configuration compatible with ARMv5
- the ARMv7 alignment configuration that supports unaligned loads and stores of 16-bit halfwords and 32-bit words.

The alignment configuration is controlled by the SCTLR.U bit, see *c1, System Control Register (SCTLR)* on page AppxG-34:

SCTLR.U == 0

ARMv5 compatible alignment support, see *Alignment* on page AppxH-6, except for the LDRD and STRD instructions. LDRD and STRD must be doubleword-aligned, otherwise:

- if SCTLR.A == 0, the instruction is UNPREDICTABLE
- if SCTLR.A == 1, the instruction causes an Alignment fault.

Note

The behavior of LDRD and STRD with SCTLR.A == 0 is compatible with ARMv5. When SCTLR.A == 1, whether the alignment check is for word or doubleword alignment is:

- IMPLEMENTATION DEFINED in ARMv5
- required to be for doubleword alignment in ARMv6.

SCTLR.U == 1

Unaligned access support for loads and stores of single 16-bit halfwords and 32-bit words, using the LDR, LDRH, LDRHT, LDRSH, LDRSHT, LDRT, STRH, STRHT, STR, and STRT instructions. Some of these instructions were introduced in ARMv6T2.

The following requirements also apply:

- LDREX and STREX exclusive access instructions must be word-aligned, otherwise the instruction generates an abort.
- In ARMv6K, an abort is generated if:
  - an LDREXH or STREXH exclusive access instruction is not halfword-aligned
  - an LDREXD or STREXD exclusive access instruction is not doubleword-aligned.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.3 アプリケーションレベルのメモリサポート

メモリサポートには、アドレスアライメント、エンディアンサポート、セマフォサポート、メモリオーダ モデル、キャッシュ、ライトバッファが含まれます。次に示すセクションでは、ARMv6 メモリサポートについてアプリケーションレベルで説明します。

- *アライメント*
- *エンディアンのサポート* P. 付録 G-7
- *セマフォのサポート* P. 付録 G-8
- *メモリモデルとメモリ順序* P. 付録 G-8

G.3.1 アライメント

ARMv6 では、次の構成がサポートされます。

- ARMv5 と互換性のある過去のアライメント構成
- 16 ビットハーフワードおよび 32 ビットワードのアンアラインドロード / ストアをサポートする ARMv7 のアライメント構成

アライメント構成は、SCTLR.U ビットにより制御されます。*c1、システム制御レジスタ (SCTLR)* P. 付録 G-34 を参照して下さい。

SCTLR.U == 0

ARMv5 互換のアライメントサポート（*アライメント* P. 付録 H-6 を参照。ただし、LDRD および STRD 命令の場合を除く）。LDRD および STRD は、ダブルワードアラインドの必要があります。そうでない場合、次の規則が適用されます。

- SCTLR.A == 0 の場合、命令は予測不能です。
- SCTLR.A == 1 の場合、命令はアライメントフォールトを引き起こします。

注

SCTLR.A == 0 での LDRD および STRD の動作は、ARMv5 と互換性があります。SCTLR.A == 1 の場合、アライメントチェックの対象がワード とダブルワードのどちらのアライメントであるかは、次のように決定されます。

- 実装定義（ARMv5 の場合）
- ダブルワードアライメントが必須（ARMv6 の場合）

SCTLR.U == 1

LDR、LDRH、LDRHT、LDRSH、LDRSHT、LDRT、STRH、STRHT、STR、STRT 命令を使用した、単精度 16 ビットハーフワードおよび 32 ビットワードのロード / ストアに対するアンアラインドアクセスのサポート。これらの命令の一部は、ARMv6T2 で導入されています。

次の要件も適用されます。

- LDREX および STREX 排他アクセス命令は、ワードアラインドの必要があります。そうでない場合、命令はアボートを生成します。
- ARMv6K では、次の場合にアボートが生成されます。
  - LDREXH または STREXH 排他アクセス命令がハーフワードアラインドでない場合
  - LDREXD または STREXD 排他アクセス命令がダブルワードアラインドでない場合

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- SWP must be word-aligned, otherwise the instruction generates an abort. From ARMv6, use of the SWP instruction is deprecated.
- All multi-word load/store instructions must be word-aligned, otherwise the instruction generates an abort.
- Unaligned access support only applies to Normal memory. Unaligned accesses to Strongly-ordered or Device memory are UNPREDICTABLE.

In both configurations, setting the SCTLR.A bit forces an abort on an unaligned access.

Note

In ARMv7, SCTLR.U is always set to 1. ARMv7 alignment support is the same as ARMv6K in this configuration.

In common with ARMv7, all instruction fetches must be aligned.

G.3.2 Endian support

ARMv6 supports the same *Big Endian* (BE) and *Little Endian* (LE) support model as ARMv7, see *Endian support* on page A3-7. It is IMPLEMENTATION DEFINED if the legacy big endian model (BE-32) defined for ARMv4 and ARMv5 is also supported. For more information about BE-32 see *Endian support* on page AppxH-7.

For configuration and control information, see *Endian configuration and control* on page AppxG-20.

BE-32 DBGWCR Byte address select values

Using the BE-32 endian model changes the meaning of the Byte address select values in DBGWCR[8:5], described in *Watchpoint Control Registers (DBGWCR)* on page C10-61. When using BE-32 endianness, use Table G-1 to interpret these values. Do not use Table C10-13 on page C10-65.

Table G-1 Byte address select values, word-aligned address, ARMv6 BE-32 endianness

DBGWCR[8:5] value	Description
0000	Watchpoint never hits
xxx1	Watchpoint hits if byte at address DBGWVR<31:2>:'11' is accessed
xx1x	Watchpoint hits if byte at address DBGWVR<31:2>:'10' is accessed
x1xx	Watchpoint hits if byte at address DBGWVR<31:2>:'01' is accessed
1xxx	Watchpoint hits if byte at address DBGWVR<31:2>:'00' is accessed

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- SWP はワードアラインドの必要があります。そうでない場合、命令はアボートを生成します。ARMv6 およびそれ以降では、SWP 命令の使用は推奨されません。
- マルチワードのロード / ストア命令は、すべてワードアラインドの必要があります。そうでない場合、命令はアボートを生成します。
- アンアラインドアクセスのサポートは、ノーマルメモリのみに適用されます。ストロングリオーダまたはデバイスメモリへのアンアラインドアクセスは予測不能です。

どちらの構成でも、SCTLR.A ビットをセットすると、アンアラインドアクセスの際に強制的にアボートが生成されます。

注

ARMv7 では、SCTLR.U は常に 1 にセットされます。この構成では、ARMv7 のアライメントサポートは ARMv6K の場合と同じです。

ARMv7 の場合と同様に、すべての命令フェッチがアラインされる必要があります。

G.3.2 エンディアンのサポート

ARMv6 では、ARMv7 と同じ *ビッグエンディアン* (BE) および *リトルエンディアン* (LE) サポートモデルがサポートされます。*エンディアンのサポート* P. A3-7 を参照して下さい。ARMv4 および ARMv5 用に定義された過去のビッグエンディアン モデル (BE-32) もサポートされるかどうかは実装定義です。BE-32 の詳細については、*エンディアンのサポート* P. 付録 H-7 を参照して下さい。

構成および制御の情報については、*エンディアン構成と制御* P. 付録 G-20 を参照して下さい。

BE-32 DBGWCR のバイトアドレス選択の値

ウォッチポイント制御レジスタ (DBGWCR) P. C10-61 で説明されているように、BE-32 エンディアンモデルを使用すると、DBGWCR[8:5] のバイトアドレス選択値の意味が変更されます。BE-32 エンディアン形式を使用する場合は、表 G-1 を使用してこれらの値を解釈します。P. C10-65 の表 C10-13 は使用しないで下さい。

表 G-1 バイトアドレス選択値、ワードアラインドアドレス、ARMv6 BE-32 エンディアン形式

DBGWCR [8:5] の値	説明
0000	ウォッチポイントは一切ヒットしません。
xxx1	アドレス DBGWVR<31:2>:'11' のバイトにアクセスするとウォッチポイントがヒットします。
xx1x	アドレス DBGWVR<31:2>:'10' のバイトにアクセスするとウォッチポイントがヒットします。
x1xx	アドレス DBGWVR<31:2>:'01' のバイトにアクセスするとウォッチポイントがヒットします。
1xxx	アドレス DBGWVR<31:2>:'00' のバイトにアクセスするとウォッチポイントがヒットします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.3.3 Semaphore support

ARM deprecates the use of the ARM semaphore instructions `SWP` and `SWPB`, in favour of the exclusive access mechanism described in *Synchronization and semaphores* on page A3-12:

- ARMv6 and ARMv6T2 support the `LDREX` and `STREX` instructions
- ARMv6K and ARMv7 add the `CLREX`, `LDREXB`, `LDREXD`, `LDREXH`, `STREXB`, `STREXD`, and `STREXH` instructions.

All Load-Exclusive and Store-Exclusive access instructions must be naturally aligned. An unaligned Exclusive access instruction generates an unaligned access Data Abort exception.

G.3.4 Memory model and memory ordering

The memory model was formalized in ARMv6. This included:

- defining Normal, Device, and Strongly-ordered memory types
- adding a Shareable memory attribute
- extending the memory attributes to support two cache policies, associated with Inner and Outer levels of cache and including a write allocation hint capability
- adding *Data Memory Barrier* (DMB) and *Data Synchronization Barrier* (DSB) operations, to support the formalized memory ordering requirements
- adding an *Instruction Synchronization Barrier* (ISB) operation, to guarantee that instructions complete before any instructions that come after them in program order are executed.

ARMv6 provided barrier operations as CP15 c7 operations. These migrated to the ARM and Thumb instruction sets as follows:

- ARMv6 required DMB, DSB, and ISB operations in CP15, see *c7, Miscellaneous functions* on page AppxH-51. The functionality of these operations is the same as that described for ARMv7 in *Memory barriers* on page A3-47.
- ARMv7 adds `DMB`, `DSB`, and `ISB` instructions to the ARM and Thumb instruction sets.

ARM deprecates use of the CP15 barrier operations.

Ordering of instructions that change the CPSR interrupt masks

In ARMv6, any instruction that implicitly or explicitly changes the interrupt masks in the CPSR and appears in program order after a Strongly-ordered access must wait for the Strongly-ordered memory access to complete. These instructions are:

- An `MSR` with the control field mask bit set.
- The flag-setting variants of arithmetic and logical instructions with the PC as the destination register. These instructions copy the SPSR to CPSR.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.3.3 セマフォのサポート

ARM セマフォ命令 `SWP` および `SWPB` の使用はお勧めしません。*同期化とセマフォ* P. A3-12 に説明されている排他アクセス機構を使用して下さい。

- ARMv6 および ARMv6T2 では、`LDREX` および `STREX` 命令がサポートされます。
- ARMv6K および ARMv7 では、`CLREX`、`LDREXB`、`LDREXD`、`LDREXH`、`STREXB`、`STREXD`、`STREXH` 命令が追加されています。

すべての排他ロードおよび排他ストアアクセス命令は、自然にアラインされる必要があります。アンアラインド排他アクセス命令では、アンアラインドアクセス データアボート例外が生成されます。

G.3.4 メモリモデルとメモリ順序

メモリモデルは ARMv6 で制式化されています。これには、次の要素が含まれます。

- ノーマル、デバイス、ストロングリオーダのメモリタイプの定義
- 共有可能メモリ属性の追加
- キャッシュの内部および外部レベルに関連し、書き込み割り当てヒント機能を含む、2 つのキャッシュポリシーをサポートするためのメモリ属性の拡張
- 制式化されたメモリ順序要件をサポートするための*データメモリ バリア*(DMB) および*データ同期バリア*(DSB) 操作の追加
- プログラムの順序で後に配置される命令が実行される前に命令が完了することを保証するための、*命令同期バリア*(ISB) 操作の追加

ARMv6 では、CP15 c7 操作としてバリア操作が提供されています。これらは、次のように ARM および Thumb 命令セットに移行されています。

- ARMv6 では、CP15 での DMB、DSB、ISB 操作が必要です。*c7、その他の機能* P. 付録 H-51 を参照して下さい。これらの操作の機能は、ARMv7 用に*メモリバリア* P. A3-47 で説明されている機能と同じです。
- ARMv7 では、ARM および Thumb 命令セットに `DMB`、`DSB`、`ISB` 命令が追加されています。

CP15 バリア操作の使用は非推奨です。

CPSR 割り込みマスクを変更する命令の順序付け

ARMv6 では、CPSR で暗黙的または明示的に割り込みマスクを変更し、プログラムの順序でストロングリオーダ アクセスの後に出現する命令は、ストロングリオーダ メモリアクセスの完了を待つ必要があります。このような命令には次のものがあります。

- 制御フィールド マスクビットがセットされている `MSR`。
- デスティネーションレジスタに PC を指定した算術および論理命令の、フラグをセットするバリアント。これらの命令は、SPSR を CPSR にコピーします。

この動作に依存することは非推奨です。ARMv7 およびそれ以降では、この動作は廃止されています。代わりに、同期が必要な場合は、メモリアクセスと次の命令との間に明示的なメモリバリアを含めます。*データ同期バリア (DSB)* P. A3-49 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARM deprecates any reliance on this behavior, and this behavior is obsolete from ARMv7. Instead, when synchronization is required, include an explicit memory barrier between the memory access and the following instruction, see *Data Synchronization Barrier (DSB)* on page A3-49.

Caches and write buffers

For details of cache support in ARMv6, see *Cache support* on page AppxG-21.

Tightly Coupled Memory (TCM) support

TCM provides low latency memory that the processor can use without the unpredictability of caches. TCM can hold critical routines, scratchpad data, or data types with locality properties that are not suitable for caching. An implementation can use TCM at the application or at the system level. For more information about ARMv6 TCM support see *Tightly Coupled Memory (TCM) support* on page AppxG-23.

DMA support

Direct Memory Access (DMA) enables a peripheral to read and write data directly from and to main memory. In ARMv6, the coherency of DMA and processor memory accesses is IMPLEMENTATION DEFINED. DMA support for TCM is IMPLEMENTATION DEFINED.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

キャッシュとライトバッファ

ARMv6 でのキャッシュサポートの詳細については、*キャッシュのサポート* P. 付録 G-21 を参照して下さい。

密結合メモリ (TCM) のサポート

TCM は、キャッシュの予測不能性なしにプロセッサが使用できる低レイテンシのメモリです。TCM には、重要なルーチン、スクラッチパッドデータ、またはキャッシュ処理に適さない局所性プロパティを持つデータタイプを保持できます。実装では、アプリケーションレベルまたはシステムレベルで TCM を使用できます。ARMv6 の TCM サポートの詳細については、*密結合メモリ (TCM) のサポート* P. 付録 G-23 を参照して下さい。

DMA のサポート

ダイレクト メモリアクセス(DMA) を使用すると、ペリフェラルはメインメモリとの間で直接データの読み出しおよび書き込みを行うことができます。ARMv6 では、DMA およびプロセッサ メモリアクセスのコヒーレンシは実装定義です。TCM における DMA のサポートは実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.4 Instruction set support

Two instruction sets are supported in ARMv6:

- the ARM instruction set
- the Thumb instruction set.

ARMv6 floating-point support, known as VFPv2, is the same as that supported in ARMv5. The instructions use coprocessors 10 and 11 and are documented with all other instructions in *Alphabetical list of instructions* on page A8-14. The following VFP instructions are not supported in ARMv6. These instructions are introduced in ARMv7 (VFPv3):

- `VMOV` (immediate)
- `VCVT` (between floating-point and fixed-point).

Note

- VFP instruction mnemonics traditionally started with an F. However this has been changed to a V prefix in the Unified Assembler Language introduced in ARMv6T2, and in many cases the rest of the mnemonic has been changed to be more compatible with other instructions mnemonics. This aligns the scalar floating-point support with the ARMv7 Advanced SIMD support, which shares some load/store and move operations to a common register file.
- The VFPv2 instructions are summarized in *F* (former VFP instruction mnemonics)* on page A8-100. This includes the two deprecated instructions in VFPv2 that do not have UAL mnemonics, the `FLDMX` and `FSTMX` instructions.

ARMv6 introduces new instructions in addition to supporting all the ARM and Thumb instructions available in ARMv5TEJ. For more information, see *Instruction set support* on page AppxH-11, *ARM instruction set support* on page AppxG-11, and *Thumb instruction set support* on page AppxG-14.

The ARM and Thumb instruction sets grew significantly in ARMv6 and ARMv6T2, compared with ARMv5TEJ, mainly because of:

- the development of ARMv6 SIMD
- the addition of many 32-bit Thumb instructions in ARMv6T2.

ARMv6K adds some kernel support instructions. It also permits the use of the optional Security Extensions and the `SMC` instruction.

ARMv7 extends the instruction sets as defined for ARMv6 and the ARMv6 architecture variants and extensions as follows:

- the introduction of barrier instructions to the ARM and Thumb instruction sets
- the ThumbEE extension in ARMv7
- the new instructions added in VFPv3
- the Advanced SIMD extension in ARMv7.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.4 命令セットのサポート

ARMv6 では、次の 2 つの命令セットがサポートされています。

- ARM 命令セット
- Thumb 命令セット

ARMv6 の浮動小数点サポートは VFPv2 と呼ばれ、ARMv5 でサポートされているものと同じです。これらの命令ではコプロセッサ 10 および 11 が使用され、他のすべての命令とともに*命令のアルファベット順リスト* P. A8-14 に記載されています。ARMv6 では、次の VFP 命令はサポートされていません。これらの命令は、ARMv7(VFPv3) で導入されたものです。

- `VMOV`（イミディエート）
- `VCVT`（浮動小数点と固定小数点との間）

注

- VFP 命令のニーモニックは従来 F で始まっていましたが、ARMv6T2 で導入された統一アセンブラ言語では V で始まるように変更されました。また、それ以外のニーモニックは多くの場合、他の命令ニーモニックとの互換性が高まるように変更されています。これにより、スカラ浮動小数点サポートと、ARMv7 のアドバンスト SIMD サポートが調整され、一般的なレジスタファイルに対して一部のロード / ストアおよび移動操作が共有されます。
- VFPv2 命令は *F* (以前の VFP 命令ニーモニック)* P. A8-100 に要約されています。ここには、UAL ニーモニックを持たない、VFPv2 で非推奨の 2 つの命令である `FLDMX` および `FSTMX` 命令が含まれています。

ARMv6 では、ARMv5TEJ で使用可能なすべての ARM 命令および Thumb 命令に加えて、新しい命令が導入されています。詳細については、*命令セットのサポート* P. 付録 H-11、*ARM 命令セットのサポート* P. 付録 G-11、*Thumb 命令セットのサポート* P. 付録 G-14 を参照して下さい。

ARMv6 および ARMv6T2 の ARM 命令セットおよび Thumb 命令セットは、ARMv5TEJ と比較して大幅に増大しています。これは、主に次のような理由によるものです。

- ARMv6 SIMD の開発
- ARMv6T2 で多数の 32 ビット Thumb 命令が追加された

ARMv6K では、いくつかのカーネルサポート命令が追加されています。さらに、オプションのセキュリティ拡張機能および `SMC` 命令も使用できます。

ARMv7 では、ARMv6 および ARMv6 アーキテクチャバリアントと拡張機能用に定義された命令セットが、次のように拡張されています。

- ARM および Thumb 命令セットにバリア命令が導入されました。
- ARMv7 で ThumbEE 拡張機能が追加されました。
- VFPv3 で新規の命令が追加されました。
- ARMv7 でアドバンスト SIMD 拡張機能が追加されました。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

This appendix describes the instructions included as a mnemonic in ARMv6. For any mnemonic, to determine which associated instruction encodings appear in a particular architecture variant, see the subsections of *Alphabetical list of instructions* on page A8-14 that describe the mnemonic. Each encoding diagram shows the architecture variants or extensions that include the encoding.

The following sections give more information about ARMv6 instruction set support:

- *ARM instruction set support*
- *Thumb instruction set support* on page AppxG-14
- *System level instruction set support* on page AppxG-14.

G.4.1 ARM instruction set support

ARMv6 includes all the ARM instructions present in ARMv5TEJ, see *ARM instruction set support* on page AppxH-12. Table G-2 shows the ARM instruction changes in the ARMv6 base architecture.

Table G-2 ARM instruction changes in ARMv6

Instruction	ARMv6 change
CPS	Introduced
LDREX	Introduced
MCRR2	Introduced
MRRC2	Introduced
PKH	Introduced
QADD16	Introduced
QADD8	Introduced
QASX	Introduced
QSUB16	Introduced
QSUB8	Introduced
QSAX	Introduced
REV, REV16, REVSH	Introduced
RFE	Introduced
SADD8, SADD16, SASX	Introduced
SEL	Introduced

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

この付録では、ARMv6 にニーモニックとして含まれている命令について説明します。いずれのニーモニックの場合も、特定のアーキテクチャバリアントに出現する関連命令エンコードを特定するには、そのニーモニックについて記述している*命令のアルファベット順リスト* P. A8-14 のサブセクションを参照して下さい。それぞれのエンコード図には、エンコードを含むアーキテクチャバリアントまたは拡張機能が示されています。

次に示すセクションでは、ARMv6 命令セットのサポートについて詳しく説明します。

- *ARM 命令セットのサポート*
- *Thumb 命令セットのサポート* P. 付録 G-14
- *システムレベルの命令セットのサポート* P. 付録 G-14

G.4.1 ARM 命令セットのサポート

ARMv6 には、ARMv5TEJ に搭載されているすべての ARM 命令が含まれています。*ARM 命令セットのサポート* P. 付録 H-12 を参照して下さい。ARMv6 基本アーキテクチャでの ARM 命令の変更点を、表 G-2 に示します。

表 G-2 ARMv6 での ARM 命令の変更点

命令	ARMv6 での変更点
CPS	導入
LDREX	導入
MCRR2	導入
MRRC2	導入
PKH	導入
QADD16	導入
QADD8	導入
QASX	導入
QSUB16	導入
QSUB8	導入
QSAX	導入
REV、REV16、REVSH	導入
RFE	導入
SADD8、SADD16、SASX	導入

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table G-2 ARM instruction changes in ARMv6 (continued)

Instruction	ARMv6 change
SETEND	Introduced
SHADD8, SHADD16	Introduced
SHSUB8, SHSUB16	Introduced
SMLAD	Introduced
SMLALD	Introduced
SMLSD	Introduced
SMLSLD	Introduced
SMMLA	Introduced
SMMLS	Introduced
SMMUL	Introduced
SMUAD	Introduced
SMUSD	Introduced
SRS	Introduced
SSAT, SSAT16	Introduced
SSUB8, SSUB16, SSAX	Introduced
STREX	Introduced
SWP	Deprecated
SWPB	Deprecated
SXTAB, SXTAB16, SXTAH	Introduced
SXTB, SXTB16, SXTH	Introduced
UADD8, UADD16, UASX	Introduced
UHADD8, UHADD16, UHASX	Introduced
UHSUB8, UHSUB16, UHSAX	Introduced
UMAAL	Introduced
UQADD8, UQADD16, UQASX	Introduced

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 G-2 ARMv6 での ARM 命令の変更点 （続き）

命令	ARMv6 での変更点
SEL	導入
SETEND	導入
SHADD8、SHADD16	導入
SHSUB8、SHSUB16	導入
SMLAD	導入
SMLALD	導入
SMLSD	導入
SMLSLD	導入
SMMLA	導入
SMMLS	導入
SMMUL	導入
SMUAD	導入
SMUSD	導入
SRS	導入
SSAT、SSAT16	導入
SSUB8、SSUB16、SSAX	導入
STREX	導入
SWP	非推奨
SWPB	非推奨
SXTAB、SXTAB16、SXTAH	導入
SXTB、SXTB16、SXTH	導入
UADD8、UADD16、UASX	導入
UHADD8、UHADD16、UHASX	導入
UHSUB8、UHSUB16、UHSAX	導入
UMAAL	導入

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table G-2 ARM instruction changes in ARMv6 (continued)

Instruction	ARMv6 change
UQSUB8, UQSUB16, UQSAX	Introduced
USAD8, USADA8	Introduced
USAT, USAT16	Introduced
USUB8, USUB16, USAX	Introduced
UXTAB, UXTAB16, UXTAH	Introduced
UXTB, UXTB16, UXTH	Introduced

The SMC instruction is added as part of the Security Extensions.

The CLREX, LDREXB, LDREXD, LDREXH, NOP, SEV, STREXB, STREXD, STREXH, WFE, WFI, and YIELD instructions are added with the enhanced kernel support as part of ARMv6K.

New ARM instructions in ARMv6T2

ARMv6T2 adds the following ARM instructions:

BFC, BFI, LDRHT, LDRSBT, LDRSHT, MLS, MOVT, RBIT, SBFX, STRHT, and UBFX.

Instructions that are only in the ARM instruction set in ARMv6T2

The following ARM instructions have no Thumb equivalents in ARMv6T2:

- register-shifted forms of the ADC, ADD, AND, BIC, CMN, CMP, EOR, MVN, ORR, RSB, SBC, SUB, TEQ, and TST instructions
- all forms of the RSC instruction
- LDMDA, LDMIB, STMDA, and STMIB
- SWP and SWPB.

ARM instructions introduced in ARMv7

The DMB, DSB, ISB, PLI, SDIV, and UDIV instructions are added in ARMv7 and are not present in any form in ARMv6. The SDIV and UDIV instructions are not present in ARMv7-A.

The DBG hint instruction is added in ARMv7. It is UNDEFINED in the ARMv6 base architecture, and executes as a NOP instruction in ARMv6K and ARMv6T2.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 G-2 ARMv6 での ARM 命令の変更点 （続き）

命令	ARMv6 での変更点
UQADD8、UQADD16、UQASX	導入
UQSUB8、UQSUB16、UQSAX	導入
USAD8、USADA8	導入
USAT、USAT16	導入
USUB8、USUB16、USAX	導入
UXTAB、UXTAB16、UXTAH	導入
UXTB、UXTB16、UXTH	導入

SMC 命令は、セキュリティ拡張機能の一部として追加されています。

CLREX、LDREXB、LDREXD、LDREXH、NOP、SEV、STREXB、STREXD、STREXH、WFE、WFI、YIELD 命令は、ARMv6K の一部として拡張カーネルサポートで追加されています。

ARMv6T2 での新規 ARM 命令

ARMv6T2 には、次の ARM 命令が追加されています。

BFC、BFI、LDRHT、LDRSBT、LDRSHT、MLS、MOVT、RBIT、SBFX、STRHT、UBFX

ARMv6T2 の ARM 命令セットにのみ存在する命令

ARMv6T2 には、次の ARM 命令に相当する Thumb 命令は存在しません。

- レジスタシフト形式の ADC、ADD、AND、BIC、CMN、CMP、EOR、MVN、ORR、RSB、SBC、SUB、TEQ、TST 命令
- すべての形式の RSC 命令
- LDMDA、LDMIB、STMDA、STMIB
- SWP および SWPB

ARMv7 で導入された ARM 命令

DMB、DSB、ISB、PLI、SDIV、UDIV 命令は ARMv7 で追加されたもので、ARMv6 にはどの形式でも存在しません。SDIV および UDIV 命令は、ARMv7-A には存在しません。

DBG ヒント命令は、ARMv7 で追加されたものです。この命令は ARMv6 基本アーキテクチャでは未定義で、ARMv6K および ARMv6T2 では NOP 命令として実行されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.4.2 Thumb instruction set support

ARMv6 includes all the Thumb instructions present in ARMv5TE, see *Thumb instruction set support* on page AppxH-15. the 16-bit Thumb instructions added in the ARMv6 base architecture are:

- CPS
- CPY
- REV, REV16, REVSH
- SETEND
- SXTB, SXTH
- UXTB, UXTH.

Thumb instruction set and ARMv6T2

From the ARMv6T2 version of the Thumb instruction set:

- The Thumb instruction set provides 16-bit and 32-bit instructions that are executed in Thumb state.
- Most forms of ARM instructions have an equivalent Thumb encoding. *Instructions that are only in the ARM instruction set in ARMv6T2* on page AppxG-13 lists the exceptions to this in ARMv6T2.

The CBZ, CBNZ, and IT instructions are only in the Thumb instruction set and are introduced in ARMv6T2.

Before ARMv6T2, a BL or BLX (immediate) Thumb instruction can be executed as a pair of 16-bit instructions, rather than as a single 32-bit instruction. For more information, see *BL and BLX (immediate) instructions, before ARMv6T2* on page AppxG-4. From ARMv6T2 these instructions are always executed as a single 32-bit instruction.

From ARMv6T2, the branch range of the BL and BLX (immediate) instructions is increased from approximately ±4MB to approximately ±16MB.

Thumb instructions introduced in ARMv7

The CLREX, LDREXB, LDREXD, LDREXH, STREXB, STREXD, and STREXH instructions are added to the Thumb instruction set in ARMv7. They are Thumb equivalents to the ARM instructions added in ARMv6K. These instructions are UNDEFINED in ARMv6T2.

The DBG, SEV, WFE, WFI, and YIELD hint instructions are added in ARMv7. They execute as NOP instructions in ARMv6T2. The 16-bit encodings of the SEV, WFE, WFI, and YIELD instructions are UNDEFINED in the ARMv6 base architecture and in ARMv6K.

G.4.3 System level instruction set support

The system instructions supported in ARMv6 are the same as those listed for ARMv7 in *Alphabetical list of instructions* on page B6-2:

- the SMC instruction only applies to the Security Extensions
- the VMRS and VMSR instructions only apply to VFP.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.4.2 Thumb 命令セットのサポート

ARMv6 には、ARMv5TE に存在するすべての Thumb 命令が含まれています。*Thumb 命令セットのサポート* P. 付録 H-15 を参照して下さい。ARMv6 基本アーキテクチャで追加された 16 ビット Thumb 命令は、次のとおりです。

- CPS
- CPY
- REV、REV16、REVSH
- SETEND
- SXTB、SXTH
- UXTB、UXTH

Thumb 命令セットと ARMv6T2

Thumb 命令セットの ARMv6T2 バージョンおよびそれ以降には、次の命令が存在します。

- Thumb 命令セットには、Thumb 状態で実行される 16 ビットおよび 32 ビット命令が用意されています。
- 大部分の形式の ARM 命令には、等価な Thumb エンコードが用意されています。ARMv6T2 での例外を *ARMv6T2 の ARM 命令セットにのみ存在する命令* P. 付録 G-13 に示します。

CBZ、CBNZ、IT 命令は Thumb 命令セットにのみ存在し、ARMv6T2 で導入されたものです。

ARMv6T2 以前では、BL または BLX（イミディエート）Thumb 命令は、単一の 32 ビット命令としてではなく、16 ビット命令のペアとして実行することができます。詳細については、*ARM v 6T2 以前の BL および BLX（イミディエート）命令* P. 付録 G-4 を参照して下さい。ARMv6T2 およびそれ以降では、これらの命令は常に単一の 32 ビット命令として実行されます。

ARMv6T2 およびそれ以降では、BL および BLX（イミディエート）命令の分岐範囲は、約 ±4MB から約 ±16MB に拡大されています。

ARMv7 で導入された Thumb 命令

ARMv7 では、Thumb 命令セットに CLREX、LDREXB、LDREXD、LDREXH、STREXB、STREXD、STREXH 命令が追加されています。これらの命令は、ARMv6K で追加された ARM 命令と等価な Thumb 命令です。これらの命令は、ARMv6T2 では未定義です。

ARMv7 には、DBG、SEV、WFE、WFI、YIELD ヒント命令が追加されています。これらの命令は、ARMv6T2 では NOP 命令として実行されます。ARMv6 基本アーキテクチャおよび ARMv6K では、SEV、WFE、WFI、YIELD 命令の 16 ビットエンコードは未定義です。

G.4.3 システムレベルの命令セットのサポート

ARMv6 でサポートされているシステム命令は、*命令のアルファベット順リスト* P. B6-2 にリストされている ARMv7 用の命令と同じです。

- SMC 命令は、セキュリティ拡張機能にのみ適用されます。
- VMRS および VMSR 命令は、VFP にのみ適用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.4.4 Different definition of some LDM and POP instructions

This difference applies to:

- LDM instructions that have the base register in the register list and specify base register writeback
- POP instructions that load at least two registers, including the base register SP.

In ARMv6, ARM instructions of these types made the value of the base register UNKNOWN, and Thumb instructions of these types were UNPREDICTABLE. Use of ARM instructions of these types is deprecated.

In ARMv7, all instructions of these types are UNPREDICTABLE.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.4.4 一部の LDM および POP 命令の別の定義

この別の定義は、次の命令に適用されます。

- レジスタリストにベースレジスタを含み、ベースレジスタのライトバックを指定する LDM 命令
- ベースレジスタ SP を含む最低 2 つのレジスタをロードする POP 命令

ARMv6 では、これらのタイプの ARM 命令についてベースレジスタの値は未知で、これらのタイプの Thumb 命令は予測不能です。これらのタイプの ARM 命令を使用することは推奨されません。

ARMv7 では、これらのタイプの命令はすべて予測不能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.5 System level register support

The general registers and processor modes are the same as ARMv7, except that the Security Extensions and Monitor mode are permitted only in ARMv6K. For more information, see Figure B1-1 on page B1-9. The following sections give information about ARMv6 system level register support:

- *Program Status Registers (PSRs)*
- *The exception model* on page AppxG-18
- *Execution environment support* on page AppxG-19.

G.5.1 Program Status Registers (PSRs)

The application level programmers' model provides the Application Program Status Register, see *APSR support* on page AppxG-3. This is an application level alias for the *Current Program Status Register* (CPSR). The system level view of the CPSR extends the register, adding state that:

- is used by exceptions
- controls the processor mode.

Each of the exception modes has its own saved copy of the CPSR, the *Saved Program Status Register* (SPSR), as shown in Figure B1-1 on page B1-9. For example, the SPSR for Monitor mode is called SPSR_mon.

The Current Program Status Register (CPSR)

The CPSR holds the following processor status and control information:

- The APSR, see *APSR support* on page AppxG-3.
- The current instruction set state. See *ISETSTATE* on page A2-15, except that ThumbEE state is not supported in ARMv6.
- The current endianness, see *ENDIANSTATE* on page A2-19.
- The current processor mode.
- Interrupt and asynchronous abort disable bits.
- In ARMv6T2, the execution state bits for the Thumb If-Then instruction, see *ITSTATE* on page A2-17.

The non-APSR bits of the CPSR have defined reset values. These are shown in the `TakeReset()` pseudocode function described in *Reset* on page B1-48, except that before ARMv6T2:

- CPSR.IT[7:0] are not defined and so do not have reset values
- the reset value of CPSR.T is 0.

The rules described in *The Current Program Status Register (CPSR)* on page B1-14 about when mode changes take effect apply with the modification that the ISB can only be the ISB operation described in *c7, Miscellaneous functions* on page AppxG-44.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.5 システムレベルのレジスタサポート

セキュリティ拡張機能およびモニタモードが ARMv6K のみで許可されていることを除いて、汎用レジスタおよびプロセッサモードは ARMv7 と同じです。詳細については、P. B1-9 の図 B1-1 を参照して下さい。次に示すセクションでは、ARMv6 のシステムレベルのレジスタサポートについて説明します。

- *プログラムステータス レジスタ (PSR)*
- *例外モデル* P. 付録 G-18
- *実行環境のサポート* P. 付録 G-19

G.5.1 プログラムステータス レジスタ (PSR)

アプリケーションレベルのプログラマモデルでは、アプリケーションプログラム ステータスレジスタが提供されます。*APSR のサポート* P. 付録 G-3 を参照して下さい。このレジスタは、*カレントプログラム ステータスレジスタ* (CPSR) のアプリケーションレベルのエイリアスです。システムレベルから見た CPSR ではこのレジスタが拡張され、次の状態が追加されます。

- 例外で使用される状態
- プロセッサモードを制御する状態

P. B1-9 の図 B1-1 に示すように、例外モードごとに、CPSR と *保存プログラム ステータスレジスタ* (SPSR) の独自の保存コピーが存在します。例えば、モニタモードの SPSR は SPSR_mon という名前です。

カレントプログラム ステータスレジスタ (CPSR)

CPSR には、次のプロセッサステータスおよび制御情報が保持されます。

- APSR（*APSR のサポート* P. 付録 G-3 を参照）
- 現在の命令セット状態。*ISETSTATE* P. A2-15 を参照して下さい。ただし、ARMv6 では ThumbEE 状態はサポートされません。
- 現在のエンディアン形式（*ENDIANSTATE* P. A2-19 を参照）
- 現在のプロセッサモード
- 割り込みおよび非同期アボートのディセーブルビット
- Thumb If-Then 命令の実行状態ビット（ARMv6T2 の場合。*ITSTATE* P. A2-17 を参照）

CPSR の非 APSR ビットによってリセット値が定義されます。これらは、*リセット* P. B1-48 で説明されている `TakeReset()` 擬似コード命令関数に示されています。ただし、ARMv6T2 以前では次のような条件があります。

- CPSR.IT[7:0] は定義されていないため、リセット時の値は存在しません。
- CPSR.T のリセット時の値は 0 です。

カレントプログラム ステータスレジスタ (CPSR) P. B1-14 で説明されている、モード変更が有効になる時期に関する規則は適用されますが、ISB が *c7、その他の機能* P. 付録 G-44 で説明されている ISB 操作のみになるという変更があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The Saved Program Status Registers (SPSRs)

The SPSRs are defined as they are in ARMv7, see *The Saved Program Status Registers (SPSRs)* on page B1-15, except that the IT[7:0] bits are not implemented before ARMv6T2.

Format of the CPSR and SPSRs

The format of the CPSR and SPSRs is the same as ARMv7:

31	30	29	28	27	26 25	24	23 20	19 16	15 10	9	8	7	6	5	4 0
N	Z	C	V	Q	IT [1:0]	J	Reserved	GE[3:0]	IT[7:2]	E	A	I	F	T	M[4:0]

In ARMv6T2, the definitions and general rules for PSR bits and support of *Non-Maskable Fast Interrupts* (NMFI) are the same as ARMv7. For more information, see *Format of the CPSR and SPSRs* on page B1-16 and *Non-maskable fast interrupts* on page B1-18.

ARMv6 and ARMv6K have the following differences:

- Bits[26:25] are RAZ/WI.
- Bits[15:10] are reserved.
- The J and T bits of the CPSR must not be changed when the CPSR is written by an `MSR` instruction, or else the behavior is UNPREDICTABLE. `MSR` instructions exist only in ARM state in these architecture variants, so this is equivalent to saying the `MSR` instructions in privileged modes must treat these bits as SBZP. `MSR` instructions in User mode still ignore writes to these bits.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

保存プログラム ステータスレジスタ (SPSR)

SPSR は、ARMv7 と同様に定義されます。*保存プログラム ステータスレジスタ (SPSR)* P. B1-15 を参照して下さい。ただし、ARMv6T2 以前では、IT[7:0] ビットは実装されていません。

CPSR と SPSR のフォーマット

CPSR と SPSR のフォーマットは ARMv7 と同じで、次のとおりです。

31	30	29	28	27	26 25	24	23 20	19 16	15 10	9	8	7	6	5	4 0
N	Z	C	V	Q	IT [1:0]	J	予約	GE[3:0]	IT[7:2]	E	A	I	F	T	M[4:0]

ARMv6T2 では、PSR ビットの定義と一般的な規則、および*マスク不能高速割り込み* (NMFI) のサポートは、ARMv7 と同じです。詳細については、*CPSR と SPSR のフォーマット* P. B1-16 と*マスク不能高速割り込み* P. B1-18 を参照して下さい。

ARMv6 および ARMv6K には、次の相違点があります。

- ビット [26:25] は RAZ/WI です。
- ビット [15:10] は予約されています。
- CPSR の J ビットおよび T ビットは、`MSR` 命令で CPSR に書き込むときに変更しないことが必要です。。変更した場合、動作は予測不能です。これらのアーキテクチャバリアントでは、`MSR` 命令は ARM 状態でのみ存在するため、これは特権モードの `MSR` 命令がこれらのビットを SBZP として処理する必要があることに相当します。ユーザモードの `MSR` 命令では、これらのビットへの書き込みは無視されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.5.2 The exception model

The exception vector offsets and priorities as stated in *Offsets from exception base addresses* on page B1-31 and *Exception priority order* on page B1-33 are the same for ARMv6 and ARMv7.

See *Exception return* on page B1-38 for the definition of exception return instructions.

The ARM abort model

ARMv6 and ARMv7 use a *Base Restored Abort Model* (BRAM), as defined in *The ARM abort model* on page AppxH-20.

Exception entry

Entry to exceptions in ARMv6 is generally as described in the sections:

- *Reset* on page B1-48
- *Undefined Instruction exception* on page B1-49
- *Supervisor Call (SVC) exception* on page B1-52
- *Secure Monitor Call (SMC) exception* on page B1-53
- *Prefetch Abort exception* on page B1-54
- *Data Abort exception* on page B1-55
- *IRQ exception* on page B1-58
- *FIQ exception* on page B1-60.

These ARMv7 descriptions are modified as follows:

- pseudocode statements that set registers, bits and fields that do not exist in the ARMv6 architecture variant are ignored
- CPSR.T is set to SCTLR.TE in ARMv6T2, as described by the pseudocode, but to 0 in ARMv6 and ARMv6K.

Fault reporting

In previous ARM documentation, in descriptions of exceptions associated with memory system faults, the terms precise and imprecise are used instead of synchronous and asynchronous. For details of the terminology used to describe exceptions in ARMv7, see *Terminology for describing exceptions* on page B1-4.

ARMv6 only supports synchronous reporting of external aborts on instruction fetches and translation table walks. In ARMv7, these faults can be reported as synchronous or asynchronous aborts. Asynchronous aborts are always reported as Data Abort exceptions.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.5.2 例外モデル

例外ベースアドレスからのオフセット P. B1-31 および *例外の優先順位* P. B1-33 で説明されている例外ベクタのオフセットと優先度は、ARMv6 と ARMv7 のどちらでも同じです。

例外からの復帰命令の定義については、*例外からの復帰* P. B1-38 を参照して下さい。

ARM アボートモデル

ARMv6 および ARMv7 では、*ARM アボートモデル* P. 付録 H-20 で定義されているように、*ベース復元アボートモデル* (BRAM) が使用されます。

例外の開始

ARMv6 での例外の開始は、次のセクションで説明されているものと一般に同じです。

- *リセット* P. B1-48
- *未定義命令例外* P. B1-49
- *スーパバイザコール (SVC) 例外* P. B1-52
- *セキュア モニタコール (SMC) 例外* P. B1-53
- *プリフェッチアボート例外* P. B1-54
- *データアボート例外* P. B1-55
- *IRQ 例外* P. B1-58
- *FIQ 例外* P. B1-60

これらの ARMv7 での説明は、次のように変更されます。

- ARMv6 アーキテクチャバリアントに存在しないレジスタ、ビット、フィールドを設定する擬似コードステートメントは無視されます。
- ARMv6T2 では、擬似コードで記述されているように CPSR.T は SCTLR.TE に設定されますが、ARMv6 および ARMv6K では 0 に設定されます。

フォールト通知

以前の ARM 説明書のメモリシステム フォールトに関連する例外の説明では、同期および非同期という用語の代わりに、正確および不正確という用語が使用されています。ARMv7 で例外について説明するために使用されている用語の詳細については、*例外に関する用語* P. B1-4 を参照して下さい。

ARMv6 では、命令フェッチおよび変換テーブルウォークでの外部アボートの同期通知のみがサポートされます。ARMv7 では、これらのフォールトは同期アボートまたは非同期アボートとして報告可能です。非同期アボートは、常にデータアボート例外として報告されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Two fault status encodings are deprecated in ARMv6:

- 0b00011 was assigned as an alignment error encoding and is re-assigned as an Access Flag section fault in ARMv6K and ARMv7
- 0b01010 was assigned as an external abort encoding and is a reserved value in ARMv7.

ARMv6 and ARMv7 provide alternative alignment and synchronous external abort error encodings that are common to both versions of the architecture.

G.5.3 Execution environment support

In ARMv6, the JOSCR.CV bit is not changed on exception entry in any implementation of Jazelle.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARMv6 では、次の 2 つのフォールトステータス エンコードは推奨されません。

- 0b00011。これはアライメントエラーのエンコードとして割り当てられており、ARMv6K および ARMv7 では、アクセスフラグ セクションフォールトとして再割り当てされています。
- 0b01010。これは外部アボートのエンコードとして割り当てられており、ARMv7 では予約済みの値です。

ARMv6 および ARMv7 では、両方のバージョンのアーキテクチャに共通する、代替のアライメントおよび同期外部アボートエラーのエンコードが提供されます。

G.5.3 実行環境のサポート

ARMv6 では、Jazelle の実装における例外の開始時に JOSCR.CV ビットは変化しません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.6 System level memory model

The pseudocode listed in *Aligned memory accesses* on page B2-31 and *Unaligned memory accesses* on page B2-32 covers the alignment behavior of all architecture variants from ARMv4. ARMv6 supports two alignment models, and the SCTLR.U bit controls the alignment configuration. For more information, see *Alignment* on page AppxG-6.

Note

- ARMv4 and ARMv5 only support the SCTLR.U = 0 alignment model.
- ARMv7 only supports the SCTLR.U = 1 alignment model.

The following sections describe the system level memory model:

- *Endian configuration and control*
- *Cache support* on page AppxG-21
- *Tightly Coupled Memory (TCM) support* on page AppxG-23
- *Virtual memory support* on page AppxG-24
- *Protected Memory System Architecture (PMSA)* on page AppxG-28.

G.6.1 Endian configuration and control

Endian control and configuration is supported by two bits in the CP15 SCTlR, and a PSR flag bit:

SCTLR.B BE-32 configuration bit. This bit must be RAZ/WI when BE-32 is not supported. BE-32 is the legacy big endian model. See *Endian support* on page AppxG-7.

SCTLR.EE This bit is used to update CPSR.E on exception entry and provide endian model information for translation table walks.

CPSR.E The flag is updated on exception entry to the value of the SCTLR.EE bit. Otherwise it is controlled by the `SETEND` instruction. Writing the bit using an `MSR` instruction is deprecated in ARMv6.

Note

BE and BE-32 are mutually exclusive. When SCTLR.B is set, SCTLR.EE and CPSR.E must be clear, otherwise the endian behavior is UNPREDICTABLE.

Endian behavior can be configured on reset using the **CFGEND[1:0]** pins. Table G-3 on page AppxG-21 defines the **CFGEND[1:0]** encoding and associated configurations.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.6 システムレベルのメモリモデル

アライン メモリアクセス P. B2-31 および*アンアラインドメモリアクセス* P. B2-32 にリストされている擬似コードでは、ARMv4 およびされ以降のすべてのアーキテクチャバリアントのアライメント動作が扱われます。ARMv6 では 2 つのアライメントモデルがサポートされ、SCTLR.U ビットによりアライメント構成が制御されます。詳細については、*アライメント* P. 付録 G-6 を参照して下さい。

注

- ARMv4 および ARMv5 では、SCTLR.U = 0 のアライメントモデルのみがサポートされます。
- ARMv7 では、SCTLR.U = 1 のアライメントモデルのみがサポートされます。

次に示すセクションでは、システムレベルのメモリモデルについて説明します。

- *エンディアン構成と制御*
- *キャッシュのサポート* P. 付録 G-21
- *密結合メモリ (TCM) のサポート* P. 付録 G-23
- *仮想メモリのサポート* P. 付録 G-24
- *保護メモリシステム アーキテクチャ (PMSA)* P. 付録 G-28

G.6.1 エンディアン構成と制御

エンディアン制御と構成は、CP15 SCTlR に含まれている次の 2 つのビット、および PSR フラグビットでサポートされます。

SCTLR.B BE-32 構成ビット。このビットは、BE-32 がサポートされていない場合には RAZ/WI にする必要があります。BE-32 は、過去のビッグエンディアン モデルです。*エンディアンのサポート* P. 付録 G-7 を参照して下さい。

SCTLR.EE このビットは、例外の開始時に CPSR.E を更新し、変換テーブルウォークのエンディアンモデル情報を提供するために使用されます。

CPSR.E このフラグは、例外の開始時に SCTLR.EE ビットの値に更新されます。それ以外の場合は、`SETEND` 命令で制御されます。ARMv6 では、`MSR` 命令を使用したビットの書き込みは推奨されません。

注

BE および BE-32 は相互に排他的です。SCTLR.B がセットされている場合、SCTLR.EE および CPSR.E をクリアする必要があります。そうしない場合、エンディアン動作は予測不能です。

エンディアン動作は、リセット時に CFGEND[1:0] ピンを使用して構成できます。CFGEND[1:0] のエンコードおよび関連する構成の定義を、P. 付録 G-21 の表 G-3 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table G-3 Configuration options on reset

CFGEND[1:0]	CP15 System Control Register, SCTLR				PSR
	EE bit	U bit	A bit	B bit	E bit
00	0	0	0	0	0
01[a]	0	0	0	1	0
10	0	1	0	0	0
11	1	1	0	0	1

a. This configuration is reserved in implementations that do not support BE-32. In this case, the B bit is RAZ.

Note

When an implementation does not include the **CFGEND[1:0]** signal, a value of 0b00 is assumed.

ARMv6 does not support the static instruction endianness configuration feature described in *Instruction endianness static configuration, ARMv7-R only* on page A3-9.

G.6.2 Cache support

ARMv7 can detect and manage a multi-level cache topology. ARMv6 only detects and manages level 1 caches, and the cache type is stored in the Cache Type Register. See *c0, Cache Type Register (CTR)* on page AppxH-35.

In ARMv6, the L1 cache must appear to software to behave as follows:

- the entries in the cache do not need to be cleaned, invalidated, or cleaned and invalidated by software for different virtual to physical mappings
- for memory regions that are described in the page tables as being Cacheable, aliases to the same physical address can exist, subject to the restrictions for 4KB small pages described in *Virtual to physical translation mapping restrictions* on page AppxG-26.

Note

These requirements are different from the required ARMv7 cache behavior described in *Address mapping restrictions* on page B3-23.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 G-3 リセット時の構成オプション

CFGEND[1:0]	CP15 システム制御レジスタ、SCTLR				PSR
	EE ビット	U ビット	A ビット	B ビット	E ビット
00	0	0	0	0	0
01[a]	0	0	0	1	0
10	0	1	0	0	0
11	1	1	0	0	1

a. BE-32 がサポートされていない実装では、この構成は予約されています。この場合、B ビットは RAZ です。

注

実装に CFGEND[1:0] 信号が含まれていない場合、値は 0b00 と見なされます。

ARMv6 では、*命令のエンディアン形式の静的な構成（ARMv7-R のみ）* P. A3-9 で説明されている静的な命令エンディアン形式の構成機能はサポートされません。

G.6.2 キャッシュのサポート

ARMv7 では、複数レベルのキャッシュトポロジを検出して管理することができます。ARMv6 ではレベル 1 のキャッシュの検出と管理のみが行われ、キャッシュタイプはキャッシュタイプ レジスタに保管されます。*c0、キャッシュタイプ レジスタ (CTR)* P. 付録 H-35 を参照して下さい。

ARMv6 では、L1 キャッシュはソフトウェアから見て、次のように動作する必要があります。

- キャッシュのエントリは、別の仮想アドレスから物理アドレスへのマッピングについて、ソフトウェアによるクリーニング、無効化、またはクリーニングと無効化を行う必要はありません。
- ページテーブルでキャッシュ可として記述されているメモリ領域の場合、*仮想アドレスから物理アドレスへの変換マッピングに関する制約条件* P. 付録 G-26 で説明されている 4KB のスモールページの制約条件に従って、同じ物理アドレスへのエイリアスが存在することができます。

注

これらの要件は、*アドレスマッピングの制限* P. B3-23 で説明されている必須の ARMv7 キャッシュ動作とは異なります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARMv6 defines a standard set of cache operations for level 1 instruction, data, and unified caches. The cache operations required are:

- for an instruction cache:
  - — invalidate all entries
  - — invalidate entries by *Modified Virtual Address* (MVA)
  - — invalidate entries by set/way
- for a data cache:
  - — invalidate all entries, clean all entries
  - — invalidate entries by MVA, clean entries by MVA
  - — invalidate entries by set/way, clean entries by set/way
- for a unified cache:
  - — invalidate all entries
  - — invalidate entries by MVA, clean entries by MVA
  - — invalidate entries by set/way, clean entries by set/way

Note

In ARMv7:

- cache operations are defined as affecting the caches when the caches are disabled.
- address based cache maintenance operations are defined as affecting all memory types.

Before ARMv7 these features of the cache operations are IMPLEMENTATION DEFINED.

ARMv6 defines a number of optional cache range operations. The defined range operations are:

- for an instruction cache:
  - — invalidate range by VA
- for a data cache:
  - — invalidate range by VA
  - — clean range by VA
  - — clean and invalidate range by VA
- prefetch related operations:
  - — prefetch instruction range by VA
  - — prefetch data range by VA
  - — stop prefetch range.

For more information, see *Block transfer operations* on page AppxG-41.

CP15 also supports configuration and control of cache lockdown. For details of the CP15 cache operation and lockdown support in ARMv6, see:

- *c7, Cache operations* on page AppxG-38
- *c9, Cache lockdown support* on page AppxG-45.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ARMv6 では、レベル 1 の命令、データ、および統一キャッシュについて、キャッシュ操作の標準的なセットが定義されます。必要なキャッシュ操作は次のとおりです。

- 命令キャッシュの場合
  - – すべてのエントリの無効化
  - – *修飾仮想アドレス*(MVA) によるエントリの無効化
  - – セット / ウェイによるエントリの無効化
- データキャッシュの場合
  - – すべてのエントリの無効化、すべてのエントリのクリーニング
  - – MVA によるエントリの無効化、MVA によるエントリのクリーニング
  - – セット / ウェイによるエントリの無効化、セット / ウェイによるエントリのクリーニング
- 統一キャッシュの場合
  - – すべてのエントリの無効化
  - – MVA によるエントリの無効化、MVA によるエントリのクリーニング
  - – セット / ウェイによるエントリの無効化、セット / ウェイによるエントリのクリーニング

注

ARMv7 では、次の規則が適用されます。

- キャッシュ操作は、キャッシュの非稼動時にキャッシュに影響を与えるように定義されます。
- アドレスベースのキャッシュ保守操作は、すべてのメモリタイプに影響を与えるように定義されます。

ARMv7 以前では、キャッシュ操作のこれらの機能は実装定義です。

ARMv6 では、オプションで多数のキャッシュ範囲操作が定義されています。定義されている範囲操作は、次のとおりです。

- 命令キャッシュの場合
  - – VA による範囲の無効化
- データキャッシュの場合
  - – VA による範囲の無効化
  - – VA による範囲のクリーニング
  - – VA による範囲のクリーニングおよび無効化
- プリフェッチ関連の操作
  - – VA による命令範囲のプリフェッチ
  - – VA によるデータ範囲のプリフェッチ
  - – 範囲のプリフェッチの停止

詳細については、*ブロック転送操作* P. 付録 G-41 を参照して下さい。

CP15 では、キャッシュロックダウンの構成と制御もサポートされます。ARMv6 での CP15 キャッシュ操作とロックダウンのサポートの詳細については、次に示すセクションを参照して下さい。

- *c7*、*キャッシュ操作* P. 付録 G-38
- *c9*、*キャッシュロックダウンのサポート* P. 付録 G-45

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Cache behavior at reset

In ARMv6, all cache lines in a cache, and all cached entries associated with branch prediction support, are invalidated by a reset. This is different to the ARMv7 behavior described in *Behavior of the caches at reset* on page B2-6.

G.6.3 Tightly Coupled Memory (TCM) support

Tightly Coupled Memory (TCM) support on page AppxG-9 introduced TCMs and their use at the application level. In addition, TCMs can be used to hold critical system-level routines such as interrupt handlers, and critical data structures such as interrupt stacks. Using TCMs can avoid indeterminate cache accesses.

ARMv6 supports up to four banks of data TCM and up to four banks of instruction TCM. You must program each bank to be in a different location in the physical memory map.

ARMv6 expects TCM to be used as part of the physical memory map of the system, and not to be backed by a level of external memory with the same physical addresses. For this reason, TCM behaves differently from a cache for regions of memory that are marked as being Write-Through Cacheable. In such regions, a write to a memory locations in the TCM never causes an external write.

A particular memory location must be contained either in the TCM or in the cache, and cannot be in both. In particular, no coherency mechanisms are supported between the TCM and the cache. This means that it is important when allocating the TCM base addresses to ensure that the same address ranges are not contained in the cache.

TCM support and VMSA

TCMs are supported in ARMv6 with VMSA support. However, there are some usage restrictions.

Restriction on translation table mappings

In a VMSA implementation, the TCM must appear to be implemented as Physically-Indexed, Physically-Addressed memory. This means it must behave as follows:

- Entries in the TCM do not have to be cleaned or invalidated by software for different virtual to physical address mappings.
- Aliases to the same physical address can exist in memory regions that are held in the TCM. This means the translation table mapping restrictions for TCM are less restrictive than for cache memory. See *Virtual to physical translation mapping restrictions* on page AppxG-26 for cache memory restrictions.

Restriction on translation table attributes

In a VMSA implementation, the translation table entries that describe areas of memory that are handled by the TCM can be Cacheable or Non-cacheable, but must not be marked as Shareable. If they are marked as either Device or Strongly-ordered, or have the Shareable attribute set, the locations that are contained in the TCM are treated as being Non-shareable, Non-cacheable.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

リセット時のキャッシュの動作

ARMv6 では、リセットによりキャッシュのすべてのキャッシュライン、および分岐予測のサポートに関連するすべてのキャッシュされたエントリが無効化されます。これは、*リセット時のキャッシュの動作* P. B2-6 で説明されている ARMv7 の動作とは異なります。

G.6.3 密結合メモリ (TCM) のサポート

密結合メモリ (TCM) のサポート P. 付録 G-9 では、TCM と、そのアプリケーションレベルでの使用が導入されています。さらに、TCM は、割り込みハンドラなどの重要なシステムレベルのルーチン、および割り込みスタックなどの重要なデータ構造を保持するために使用できます。TCM を使用すると、不定キャッシュアクセスを回避することができます。

ARMv6 では、最大 4 つのデータ TCM のバンクと、最大 4 つの命令 TCM のバンクがサポートされます。各バンクは、物理メモリマップの別の位置に配置されるようにプログラムする必要があります。

ARMv6 では、TCM はシステムの物理メモリマップの一部として使用されること、および同じ物理アドレスを持つあるレベルの外部メモリにデータが重複していないことが想定されています。このため、TCM は、ライトスルーキャッシュ可としてマークされているメモリ領域用のキャッシュとは異なる動作をします。このような領域では、TCM のあるメモリ位置への書き込みにより、外部書き込みが発生することは一切ありません。

特定のメモリ位置は、TCM またはキャッシュのいずれかに含まれている必要がありますが、両方に含まれていることは許されません。特に、TCM とキャッシュとの間ではコヒーレンシ機構はサポートされていません。これは、TCM のベースアドレスの割り当て時に、キャッシュに同じアドレス範囲が含まれていないことを確認するのが重要であることを意味します。

TCM サポートと VMSA

TCM は、VMSA サポートを持つ ARMv6 でサポートされます。ただし、使用に一部制約条件があります。

変換テーブルマッピングに関する制約条件

VMSA 実装では、TCM は、物理インデクス、物理アドレス指定のメモリとして実装する必要があります。つまり、次のように動作する必要があります。

- 仮想アドレスから物理アドレスへの異なるマッピングについて、ソフトウェアで TCM のエントリのクリーニングまたは無効化を行う必要はありません。
- TCM に保持されているメモリ領域に、同じ物理アドレスへのエイリアスが存在することができます。これは、TCM の変換テーブルマッピングの制約条件がキャッシュメモリの場合よりも緩いことを意味します。キャッシュメモリの制約条件については、*仮想アドレスから物理アドレスへの変換マッピングに関する制約条件* P. 付録 G-26 を参照して下さい。

変換テーブル属性に関する制約条件

VMSA 実装では、TCM によって処理されるメモリの領域を記述する変換テーブルエントリは、キャッシュ可またはキャッシュ不可のいずれでも可能ですが、共有可能としてマークすることはできません。それらがデバイスまたはストロングリオーダのいずれかとしてマークされているか、あるいは共有可能属性セットを持つ場合、TCM に含まれている位置は共有不可、キャッシュ不可として処理されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

TCM CP15 configuration and control

In ARMv7, a TCM Type Register is required. However, its format can be compatible with ARMv6 or IMPLEMENTATION DEFINED. For more information, see *c0, TCM Type Register (TCMTR)* on page B3-85.

In ARMv6, CP15 c0 and c9 registers configure and control the TCMs in a system. For more information, see:

- *c0, TCM Type Register (TCMTR)* on page AppxG-33
- *c9, TCM support* on page AppxG-46.

Note

In addition to the basic TCM support model in ARMv6, a set of range operations that can operate on caches and TCMs are documented. Range operations are considered optional in ARMv6. See *Block transfer operations* on page AppxG-41.

The ARM Architecture Reference Manual (DDI 0100) described an ARMv6 feature known as SmartCache, and a level 1 DMA model associated with TCM support. Both of these features are considered as IMPLEMENTATION DEFINED, and are not described in this manual.

In some implementations of ARMv4 and ARMv5, bits in the CP15 System Control Register, SCTLR[19:16] or a subset, are used for TCM control. From ARMv6 these bits have fixed values, and no SCTLR bits are used for TCM control.

G.6.4 Virtual memory support

A key component of the Virtual Memory System Architecture (VMSA) is the use of translation tables. ARMv6 supports two formats of virtual memory translation table:

- a legacy format for ARMv4 and ARMv5 compatibility
- a revised format, called the VMSAv6 format, that is also used in ARMv7.

Both table formats support use of the *Fast Context Switch Extension* (FCSE), but ARM deprecates use of the FCSE, and the FCSE is optional in ARMv7. For the differences in VMSAv6 format support between ARMv6 and ARMv6K, see *VMSAv6 translation table format* on page AppxG-26.

Note

- ARMv7 does not support the legacy format.
- ARMv7 VMSA support is the same as that supported by the revised format in ARMv6K, except for the address mapping restrictions described:
  - for ARMv6 in *Virtual to physical translation mapping restrictions* on page AppxG-26
  - for ARMv7 in *Address mapping restrictions* on page B3-23.
- For more information about the FCSE see Appendix E *Fast Context Switch Extension (FCSE)*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

TCM CP15 の構成と制御

ARMv7 では、TCM タイプレジスタは必須です。ただし、そのフォーマットは ARMv6 互換、または実装定義にできます。詳細については、*c0、TCM タイプレジスタ (TCMTR)* P. B3-85 を参照して下さい。

ARMv6 では、CP15 c0 および c9 レジスタにより、システム内の TCM の構成と制御が行われます。詳細については、次のセクションを参照して下さい。

- *c0、TCM タイプレジスタ (TCMTR)* P. 付録 G-33
- *c9、TCM のサポート* P. 付録 G-46

注

ARMv6 での基本的な TCM サポートモデルとともに、キャッシュおよび TCM で動作することができる一連の範囲操作が文書化されています。ARMv6 では、範囲操作はオプションと見なされます。*ブロック転送操作* P. 付録 G-41 を参照して下さい。

『*ARM アーキテクチャ リファレンスマニュアル*』(DDI 0100) では、SmartCache と呼ばれる ARMv6 機能、および TCM サポートに関連するレベル 1 の DMA モデルについて説明されています。これらの機能は両方ともに実装定義と見なされるため、本書では説明されていません。

ARMv4 および ARMv5 の一部の実装では、CP15 システム制御レジスタのビット SCTLR[19:16]、またはそのサブセットが、TCM 制御に使用されます。ARMv6 およびそれ以降では、これらのビットの値は固定されており、SCTLR のビットは TCM 制御には使用されません。

G.6.4 仮想メモリのサポート

仮想メモリシステム アーキテクチャ (VMSA) の主要なコンポーネントの 1 つは、変換テーブルの使用です。ARMv6 では、2 つのフォーマットの仮想メモリ変換テーブルがサポートされます。

- ARMv4 および ARMv5 と互換性のある過去のフォーマット
- VMSAv6 フォーマットと呼ばれる改訂フォーマット。これは ARMv7 でも使用されます。

どちらのテーブルフォーマットでも *高速コンテキストスイッチ拡張機能* (FCSE) の使用がサポートされますが、FCSE の使用は非推奨で、ARMv7 では FCSE はオプションです。ARMv6 と ARMv6K との VMSAv6 フォーマットサポートの相違点については、*VMSAv6 の変換テーブルのフォーマット* P. 付録 G-26 を参照して下さい。

注

- ARMv7 では、過去のフォーマットはサポートされません。
- ARMv7 VMSA サポートは、ARMv6K の改訂フォーマットでのサポートと同じです。ただし、次のセクションで説明されている、アドレスマッピングの制約が存在します。
  - *仮想アドレスから物理アドレスへの変換マッピングに関する制約条件* P. 付録 G-26 (ARMv6 の場合)
  - *アドレスマッピングの制限* P. B3-23 (ARMv7 の場合)
- FCSE の詳細については、付録 E *「高速コンテキストスイッチ拡張機能 (FCSE)」*を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Execute Never (XN)

The ARMv7 requirement that instruction prefetches are not made from read-sensitive devices also applies to earlier versions of the architecture:

- ARMv7 requires you to mark all read-sensitive devices with the Execute-never (XN) to ensure that this requirement is met, see *The Execute Never (XN) attribute and instruction prefetching* on page B3-30
- before ARMv7, how this requirement is met is IMPLEMENTATION DEFINED.

Legacy translation table format

ARMv6 legacy support only includes the coarse translation table type as described in *Second level Coarse page table descriptor format* on page AppxH-25. ARMv6 does not support the fine level 2 Page table format. Therefore the legacy translation table format includes subpage access permissions but does not support 1KB Tiny pages. Table G-4 shows the legacy first level translation table entry formats.

Table G-4 Legacy first level descriptor format

31 … 20	19 … 14	12	11	10	9	8 … 5	4	3	2	1	0
Fault	IGN									0	0
Coarse page table	Coarse page table base address				IMP	Domain	SBZ			0	1
Section	Section base address	SBZ	TEX		AP	IMP	Domain	SBZ	C	B	1	0
	Reserved									1	1

Note

ARMv5TE includes optional support for Supersections, Shareable memory, and the TEX bitfield. See *Virtual memory support* on page AppxH-21.

Use of the SCTLR.S and SCTLR.R bits described in Table H-6 on page AppxH-23 is deprecated. They are implemented for use only with the legacy format translation tables, and their use is not supported in VMSAv6 or VMSAv7.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

実行不可 (XN)

命令プリフェッチは読み出しで影響を受けるデバイスからは実行されないという ARMv7 の要件は、アーキテクチャの以前のバージョンにも適用されます。

- ARMv7 では、読み出しで影響を受けるすべてのデバイスを実行不可 (XN) にマークして、この要件が満たされることを保証する必要があります（*実行不可 (XN) 属性と命令のプリフェッチ* P. B3-30 を参照）。
- ARMv7 以前では、この要件を満たす方法は実装定義です。

過去の変換テーブルフォーマット

ARMv6 の過去のサポートには、*第 2 レベルのコアース ページテーブル記述子のフォーマット* P. 付録 H-25 で説明されているコアース変換テーブルタイプのみが含まれています。ARMv6 では、ファインレベル 2 のページテーブル フォーマットはサポートされません。このため、過去の変換テーブルフォーマットにはサブページアクセス許可が含まれていますが、1KB のタイニーページはサポートされていません。過去の第 1 レベル変換テーブルエントリのフォーマットを、表 G-4 に示します。

表 G-4 過去の第 1 レベル記述子フォーマット

31 … 20	19 … 14	12	11	10	9	8 … 5	4	3	2	1	0
フォールト	IGN									0	0
コアース ページテーブル	コアース ページテーブルのベースアドレス				IMP	ドメイン	SBZ			0	1
セクション	セクションのベースアドレス	SBZ	TEX		AP	IMP	ドメイン	SBZ	C	B	1	0
	予約									1	1

注

ARMv5TE には、スーパーセクション、共有可能メモリ、および TEX ビットフィールドに対するオプションのサポートが含まれています。*仮想メモリのサポート* P. 付録 H-21 を参照して下さい。

P. 付録 H-23 の表 H-6 で説明されている SCTLR.S ビットおよび SCTLR.R ビットの使用は非推奨です。これらのビットは過去のフォーマットの変換テーブルで使用するためにのみ実装されており、VMSAv6 または VMSAv7 でこれらのビットを使用することはサポートされていません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

VMSAv6 translation table format

The VMSAv6 translation table format is fully compatible with the virtual memory support in ARMv7-A. It includes the following features:

- the ability to mark a virtual address as either global or context-specific
- the ability to encode the Normal, Device, or Strongly-ordered memory type into the translation tables
- the Shareable attribute
- the XN execute never access permission attribute
- a third AP bit
- a TEX bitfield used with the C and B bits to define the cache attributes for each page of memory
- support for an application specific (ASID) or global identifier
- 16MB Supersections, and the ability to map a Supersection to a 16MB range.

Related to this new translation table format, VMSAv6 provides:

- support for two translation table base registers and an associated control register
- independent fault status and fault address registers for reporting Prefetch Abort exceptions and Data Abort exceptions
- a Context ID Register, CONTEXTIDR.

ARMv6K added the following features to VMSAv6:

- An additional access permission encoding, AP[2:0] == 0b111, and an associated simplified access permissions model. See *Access permissions* on page B3-28, and *Simplified access permissions model* on page B3-29.
- The access flag feature. See *The access flag* on page B3-21.
- TEX remapping. See *Memory region attribute descriptions when TEX remap is enabled* on page B3-34.

Virtual to physical translation mapping restrictions

An ARMv6 implementation can restrict the mapping of pages that remap virtual address bits [13:12]. This restriction, called page coloring, supports the handling of aliases by an implementation that uses VIPT caches. On an implementation that imposes this restriction, the most significant bit of the cache size fields for the instruction and data caches in the CTR is Read-As-One, see *c0, Cache Type Register (CTR)* on page AppxH-35.

To avoid alias problems, this restriction enables these bits of the virtual address to be used to index into the cache without requiring hardware support. The restriction supports virtual indexing on caches where a cache way has a maximum size of 16KB. There is no restriction on the number of ways supported. Cache ways of

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

VMSAv6 の変換テーブルのフォーマット

VMSAv6 の変換テーブルのフォーマットは、ARMv7-A の仮想メモリサポートと完全に互換性があります。このフォーマットには、次の機能が含まれています。

- 仮想アドレスをグローバルまたはコンテキスト固有のいずれかとしてマークする機能
- ノーマル、デバイス、またはストロングリオーダ メモリタイプを変換テーブルにエンコードする機能
- 共有可能属性
- XN（実行不可）アクセス許可属性
- 3 番目の AP ビット
- メモリの各ページのキャッシュ属性を定義するために、C ビットおよび B ビットとともに使用される TEX ビットフィールド
- アプリケーション固有 ID(ASID) またはグローバル ID のサポート
- 16MB スーパーセクション、およびスーパーセクションを 16MB 範囲にマッピングする機能

この新規変換テーブルフォーマットに関連して、VMSAv6 では次のものがサポートされます。

- 2 つの変換テーブル ベースレジスタおよび関連する制御レジスタのサポート
- プリフェッチアボート例外およびデータアボート例外を報告するために使用される、独立したフォールトステータス レジスタおよびフォールトアドレス レジスタ
- コンテキスト ID レジスタ、CONTEXTIDR

ARMv6K では、VMSAv6 に次の機能が追加されています。

- 追加のアクセス許可エンコードである AP[2:0] == 0b111、および関連する単純化されたアクセス許可モデル。*アクセス許可* P. B3-28 および *単純なアクセス許可モデル* P. B3-29 を参照して下さい。
- アクセスフラグ機能。*アクセスフラグ* P. B3-21 を参照して下さい。
- TEX の再マッピング。*TEX 再マップが許可されている場合のメモリ領域の属性の説明* P. B3-34 を参照して下さい。

仮想アドレスから物理アドレスへの変換マッピングに関する制約条件

ARMv6 実装では、仮想アドレスのビット [13:12] を再マッピングするページのマッピングを制限できます。ページカラーリングと呼ばれるこの制約条件では、VIPT キャッシュを使用する実装によるエイリアスの処理がサポートされます。この制約を受ける実装においては、CTR の命令キャッシュおよびデータキャッシュのキャッシュサイズ フィールドの最上位ビットは、読み出し値が 1(RAO) です。*c0、キャッシュタイプ レジスタ (CTR)* P. 付録 H-35 を参照して下さい。

エイリアスの問題を回避するには、この制約条件を使用して、仮想アドレスのこれらのビットがハードウェアのサポートを必要とせず、キャッシュへのインデクス付けに使用されるようにします。この制約条件では、キャッシュウェイの最大サイズが 16KB である、キャッシュの仮想インデクス付けがサポートされます。サポートされるウェイの数に制限はありません。4KB 以下のキャッシュウェイは、この制限を受けません。これは、仮想アドレスまたは物理アドレスは、単一のキャッシュセットにのみ割り当てることができるためです。仮想インデクス付けに関連する ARMv6 キャッシュポリシーは、次のようになります。ここで、NSETS はセットの数、LINELEN はキャッシュライン長です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

4KB or less do not suffer from this restriction, because any address (virtual or physical) can only be assigned to a single cache set. Where NSETS is the number of sets, and LINELEN is the cache line length, the ARMv6 cache policy associated with virtual indexing is:

```
Log2(NSETS×LINELEN) =< 12       ; no VI restriction
12 < Log2(NSETS×LINELEN) =< 14  ; VI restrictions apply
Log2(NSETS×LINELEN) > 14        ; PI only, VI not supported
```

If a page is marked as Non-shareable, then if the most significant bits of the cache size fields are RAO, the implementation requires the remapping restriction and the following restrictions apply:

- If multiple virtual addresses are mapped onto the same physical addresses, then for all mappings bits [13:12] of the virtual address must be equal, and must also be equal to bits [13:12] of the physical address. The same physical address can be mapped by TLB entries of different page sizes. These can be 4KB, 64KB, or sections.
- If all mappings to a physical address are of a page size equal to 4KB, the restriction that bits [13:12] of the virtual address must equal bits [13:12] of the physical address is not required. Bits [13:12] of all virtual address aliases must still be equal.

There is no restriction on the more significant bits in the virtual address.

If a page is marked as Shareable and Cacheable, memory coherency must be maintained across the shareability domain. In ARMv7, software manages instruction coherency, and data caches must be transparent. See *Shareable, Inner Shareable, and Outer Shareable Normal memory* on page A3-30 for more information.

Note

In some implementations, marking areas of memory as Shareable can have substantial performance effects, because those areas might not be held in caches.

ARMv6 and the Security Extensions

The Security Extensions provide virtual memory support for two physical address spaces as described in *Secure and Non-secure address spaces* on page B3-26 and are supported from ARMv6K. Support is the same as in ARMv7 with the following exceptions:

- ARMv6 only supports CP15 operations for virtual to physical address translation as part of the Security Extensions. ARMv7 includes support in the base architecture. For details see *Virtual Address to Physical Address translation operations* on page B3-63.
- Additional bits are allocated in the NSACR register. See *c1, VMSA Security Extensions support* on page AppxG-35.
- When implemented, the Cache Dirty Status Register is a Banked register. See *c7, Cache Dirty Status Register (CDSR)* on page AppxG-39.
- A Cache Behavior Override Register is defined. See *c9, Cache Behavior Override Register (CBOR)* on page AppxG-49.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.


```
Log2(NSETS×LINELEN) =< 12       ; no VI restriction
12 < Log2(NSETS×LINELEN) =< 14  ; VI restrictions apply
Log2(NSETS×LINELEN) > 14        ; PI only, VI not supported
```

ページが共有不可にマークされており、キャッシュサイズ フィールドの最上位ビットが RAO の場合、実装では再マッピングの制約条件が必要になり、次の制限が適用されます。

- 複数の仮想アドレスが同じ物理アドレスにマッピングされる場合、すべてのマッピングについて仮想アドレスのビット [13:12] が一致している必要があり、さらに物理アドレスのビット [13:12] とも一致する必要があります。同じ物理アドレスを、異なるページサイズの TLB エントリによってマッピングできます。4KB、64KB、セクションのいずれかが可能です。
- 物理アドレスへのマッピングがすべて 4KB のページサイズによるものである場合、仮想アドレスのビット [13:12] が物理アドレスのビット [13:12] と同じである必要があるという制限は不要です。ただし、この場合も仮想アドレスエイリアスのビット [13:12] はすべて同じ必要があります。

仮想アドレスの他の上位ビットに関する制限はありません。

ページが共有可能でキャッシュ可としてマークされている場合は、共有可能ドメイン間でメモリコヒーレンシが維持されている必要があります。ARMv7 では、ソフトウェアにより命令コヒーレンシが管理されており、データキャッシュは透過的な必要があります。詳細については、*共有可能、内部共有可能、外部共有可能のノーマルメモリ* P. A3-30 を参照して下さい。

注

一部の実装では、メモリの領域を共有可能としてマークすると、大幅にパフォーマンスが向上する場合があります。これは、それらの領域がキャッシュに保持されない可能性があるためです。

ARMv6 とセキュリティ拡張機能

セキュリティ拡張機能では、*セキュアアドレス空間と非セキュアアドレス空間* P. B3-26 で説明されている 2 つの物理アドレス空間について仮想メモリのサポートが提供されます。これは、ARMv6K およびそれ以降でサポートされています。次の例外を除いて、サポートは ARMv7 と同じです。

- ARMv6 では、セキュリティ拡張機能の一部として、仮想アドレスから物理アドレスへの変換に使用する CP15 操作のみがサポートされます。ARMv7 には、基本アーキテクチャにサポートが組み込まれています。詳細については、*仮想アドレスから物理アドレスへの変換操作* P. B3-63 を参照して下さい。
- 追加のビットは、NSACR レジスタに割り当てられます。*c1、VMSA セキュリティ拡張機能のサポート* P. 付録 G-35 を参照して下さい。
- 実装されている場合、キャッシュダーティステータス レジスタはバンクレジスタです。*c7、キャッシュダーティステータス レジスタ (CDSR)* P. 付録 G-39 を参照して下さい。
- キャッシュ動作オーバライドレジスタが定義されます。*c9、キャッシュ動作オーバライドレジスタ (CBOR)* P. 付録 G-49 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- TCM access support registers are defined. See *c9, TCM Non-Secure Access Control Registers, DTCM-NSACR and ITCM-NSACR* on page AppxG-51.

CP15 support for the Security Extensions in ARMv7 is defined in *Effect of the Security Extensions on the CP15 registers* on page B3-71.

CP15SDISABLE input

The effect of this input is described for ARMv7 in *The CP15SDISABLE input* on page B3-76. In ARMv6K, TCM support is affected as follows:

- the DTCM_NSAC and ITM_NSAC registers are added to the controlled register list
- any TCM region registers restricted to Secure access only by the NSACR register settings are added to the controlled register list.

G.6.5 Protected Memory System Architecture (PMSA)

PMSA in ARMv5 is IMPLEMENTATION DEFINED. The method described in *Protected memory support* on page AppxH-28 is only supported in ARMv4 and ARMv5. PMSA is formalized in ARMv6 under a different CP15 support model.

The PMSA support in ARMv6 (PMSAv6) differs from PMSAv7 in the following ways:

- PMSAv6 does not support subregions as defined in *Subregions* on page B4-3.
- The default memory map shown in Table B4-1 on page B4-6 and Table B4-2 on page B4-7 does not support the XN bit for restricting instruction fetches. The affected addresses are treated as Normal, Non-cacheable in PMSAv6.
- The default memory map applies only when the MPU is disabled. The SCTLR.BR bit is not supported in PMSAv6.
- TCM memory behaves as normal when the TCM region is enabled and the MPU is disabled.

In all other respects, PMSAv6 is as described for ARMv7 in Chapter B4 *Protected Memory System Architecture (PMSA)*.

Execute Never (XN)

The ARMv7 requirement that instruction prefetches are not made from read-sensitive devices also applies to earlier versions of the architecture:

- ARMv7 requires you to mark all read-sensitive devices with the Execute-never (XN) to ensure that this requirement is met, see *The Execute Never (XN) attribute and instruction prefetching* on page B3-30
- before ARMv7, how this requirement is met is IMPLEMENTATION DEFINED.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- TCM アクセスサポート レジスタが定義されます。*c9、TCM 非セキュアアクセス制御レジスタ、DTCM-NSACR および ITCM-NSACR* P. 付録 G-51 を参照して下さい。

ARMv7 でのセキュリティ拡張機能に対する CP15 サポートは、*CP15 レジスタに対するセキュリティ拡張機能の影響* P. B3-71 で定義されています。

CP15SDISABLE 入力

ARMv7 でのこの入力の効果については、*CP15SDISABLE 入力* P. B3-76 で説明されています。ARMv6K では、TCM サポートは次のような影響を受けます。

- DTCM_NSAC および ITM_NSAC レジスタが、制御されるレジスタリストに追加されます。
- NSACR レジスタ設定によるセキュアなアクセスのみに制限されているすべての TCM 領域レジスタが、制御されるレジスタリストに追加されます。

G.6.5 保護メモリシステム アーキテクチャ (PMSA)

ARMv5 での PMSA は実装定義です。*保護メモリのサポート* P. 付録 H-28 で説明されている方法は、ARMv4 および ARMv5 でのみサポートされます。ARMv6 では、PMSA は異なる CP15 サポートモデルで制式化されています。

ARMv6 での PMSA サポート (PMSAv6) は、PMSAv7 とは次の点が異なります。

- *サブ領域* P. B4-3 で説明されているように、PMSAv6 ではサブ領域はサポートされません。
- P. B4-6 の表 B4-1 および P. B4-7 の表 B4-2 に示されているデフォルトのメモリマップでは、命令フェッチを制限する XN ビットはサポートされません。PMSAv6 では、影響を受けるアドレスは、ノーマルかつキャッシュ不可として扱われます。
- デフォルトのメモリマップは、MPU が非稼動の場合にのみ適用されます。PMSAv6 では、SCTLR.BR ビットはサポートされません。
- TCM 領域が稼動し、MPU が非稼動の場合、TCM メモリはノーマルメモリとして動作します。

他のすべての点については、PMSAv6 は第 B4 章 *「保護メモリシステム アーキテクチャ (PMSA)」* で説明されている ARMv7 の場合と同じです。

実行不可 (XN)

命令プリフェッチは読み出しで影響を受けるデバイスからは実行されないという ARMv7 の要件は、アーキテクチャの以前のバージョンにも適用されます。

- ARMv7 では、読み出しで影響を受けるすべてのデバイスを実行不可 (XN) にマークして、この要件が満たされることを保証する必要があります（*実行不可 (XN) 属性と命令のプリフェッチ* P. B3-30 を参照）。
- ARMv7 以前では、この要件を満たす方法は実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.7 System Control coprocessor (CP15) support

Much of the CP15 support is common to VMSAv6 and PMSAv6. However:

- some registers are unique to each memory system architecture
- some registers have different functionality in the two memory system architectures, for example the SCTLR.

The following sections summarize the ARMv6 implementations of the CP15 registers:

- *Organization of CP15 registers for an ARMv6 VMSA implementation*
- *Organization of CP15 registers for an ARMv6 PMSA implementation* on page AppxG-31.

The rest of this section describes the ARMv6 CP15 support in order of the CRn value. The description of each register:

- indicates if the register is unique to VMSA or PMSA
- indicates any differences between the two implementations if the register is included in both VMSA and PMSA implementations.

Note

This approach is different from that taken in Part B of this manual, where:

- *CP15 registers for a VMSA implementation* on page B3-64 is a complete description of CP15 support in a VMSAv7 implementation
- *CP15 registers for a PMSA implementation* on page B4-22 is a complete description of CP15 support in a PMSAv7 implementation.

The convention used for fixed bitfields in the CP15 register definitions is defined in *Meaning of fixed bit values in register diagrams* on page B3-78.

In ARMv6 the execution of an `MCR` or `MRC` instruction with an unallocated CP15 register encoding is `UNPREDICTABLE`.

ARMv6 provides some `MCRR` instructions to support block transfers, see *Block transfer operations* on page AppxG-41.

G.7.1 Organization of CP15 registers for an ARMv6 VMSA implementation

Figure G-1 on page AppxG-30 shows the CP15 registers in an ARMv6 VMSA implementation:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.7 システム制御コプロセッサ（CP15）のサポート

CP15 サポートの大部分は、VMSAv6 と PMSAv6 とで共通です。ただし、次の点は異なります。

- 一部のレジスタは、各メモリシステム アーキテクチャに固有です。
- 一部のレジスタ（例えば、SCTLR）は、2 つのメモリシステム アーキテクチャで機能が異なります。

次に示すセクションでは、CP15 レジスタの ARMv6 実装が要約されています。

- *ARMv6 VMSA 実装での CP15 レジスタの編成*
- *ARMv6 PMSA 実装での CP15 レジスタの編成* P. 付録 G-31

このセクションの以降の部分では、CRn 値の順に ARMv6 CP15 サポートについて説明します。各レジスタの説明では、次の内容が示されています。

- レジスタが VMSA または PMSA に固有であるかどうか
- レジスタが VMSA および PMSA 実装の両方に含まれている場合、2 つの実装間に存在する相違点

注

この手法は、本書のパート B で採用されている手法とは異なります。パート B では、次のセクションに説明があります。

- *VMSA 実装の CP15 レジスタ* P. B3-64 は、VMSAv7 実装での CP15 サポートに関する完全な説明です。
- *PMSA 実装の CP15 レジスタ* P. B4-22 は、PMSAv7 実装での CP15 サポートに関する完全な説明です。

CP15 レジスタ定義の固定ビットフィールドで使用される規則は、*レジスタ図の固定ビット値の意味* P. B3-78 で定義されています。

ARMv6 では、未割り当ての CP15 レジスタエンコードの `MCR` または `MRC` 命令を実行すると、結果は予測不能です。

ARMv6 では、ブロック転送をサポートするいくつかの `MCRR` 命令が用意されています。*ブロック転送操作* P. 付録 G-41 を参照して下さい。

G.7.1 ARMv6 VMSA 実装での CP15 レジスタの編成

ARMv6 VMSA 実装の CP15 レジスタを、P. 付録 G-30 の図 G-1 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CRn	opc1	CRm	opc2		
c0	0	c0	0		MIDR, Main ID Register
			1		CTR, Cache Type Register
			2		TCMTR, TCM Type Register
			3		TLBTR, TLB Type Register
			*5		* MPIDR, Multiprocessor Affinity Register
			{4,*5,6-7}		Aliases of Main ID Register
		*{c1-c7}	*{0-7}		*CPUID Registers, optional
c1	0	c0	{0-2}		System Control registers
		*c1	*{0-2}		† Security Extensions registers
c2	0	c0	{0-2}		Translation Table Base Registers
c3	0	c0	0		DACR, Domain Access Control Register
c5	0	c0	{0,1}		Fault Status Registers
c6	0	c0	{0-2}		Fault Address Registers
c7	0	c0	4		CP15WFI, Wait for interrupt operation
		c5	{0-2,6,7}		Cache and branch predictor maintenance operations
			4		**CP15ISB, Instruction barrier operation**
		{c6,c7}	{0-2}		Cache maintenance operations
		c10	{0-3}		Cache maintenance operations
			{4,5}		**Data barrier operations**
			6		CDSR, Cache Dirty Status Register
		c11	{0-2}		Cache maintenance operations
		c12	{4,5}	‡	**Block transfer support operations**, optional
		c13	1		Cache prefetch operation, optional
		c14	{0-3}	‡	Cache maintenance operations
c8	0	{c5-c7}	{0-2}		TLB maintenance operations
c9	0	{c0,c1}	{0,1}		Cache Lockdown (format C) and TCM Region registers
		*c1	*{2,3}		† TCM Non-secure Access Control Registers
		c2	0		TCMSR, TCM Selection Register
		{c5,c6}	{0,1}		Cache lockdown (format D) operations
		*c8	*0		† CBOR, Cache Behavior Override Register
c10	0	c0	{0,1}		TLB Lockdown Registers
		*c2	*{0,1}		* TEX remap registers (PRRR, NMRR)
		{c4,c8}	{0,1}	‡	TLB Lockdown operations and registers, optional
c11	{0-7}	{c0-c8,c15}	{0-7}	‡	Reserved for DMA support for TCM operations
c12	0	*c0	*{0,1}		† Security Extensions registers
		*c1	*0		† ISR, Interrupt Status Register
c13	0	c0	0		FCSEIDR, FCSE PID Register
			1		CONTEXTIDR, Context ID Register
			*{2-4}		*Software Thread ID registers
c15	{0-7}	{c0-c15}	{0-7}	‡	IMPLEMENTATION DEFINED registers

Read-only | Read/Write | Write-only | **Bold text = Accessible in User mode**

* Not available in all versions of ARMv6 † If Security Extensions implemented ‡ Access depends on operation

Figure G-1 CP15 registers in an ARMv6 VMSA implementation

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

CRn	opc1	CRm	opc2		
c0	0	c0	0		MIDR、メインIDレジスタ
			1		CTR、キャッシュタイプ レジスタ
			2		TCMTR、TCMタイプレジスタ
			3		TLBTR、TLBタイプレジスタ
			*5		* MPIDR、マルチプロセッサ類似性レジスタ
			{4,*5,6-7}		メインIDレジスタのエイリアス
		*{c1-c7}	*{0-7}		*CPUIDレジスタ(オプション)
c1	0	c0	{0-2}		システム制御レジスタ
		*c1	*{0-2}		† セキュリティ拡張機能レジスタ
c2	0	c0	{0-2}		変換テーブル ベースレジスタ
c3	0	c0	0		DACR、ドメインアクセス制御レジスタ
c5	0	c0	{0,1}		フォールトステータス レジスタ
c6	0	c0	{0-2}		フォ-ルトアドレス レジスタ
c7	0	c0	4		CP15WFI、割り込み待ち操作
		c5	{0-2,6,7}		キャッシュおよび分岐予測器の保守操作
			4		**CP15ISB、命令バリア操作**
		{c6,c7}	{0-2}		キャッシュ保守操作
		c10	{0-3}		キャッシュ保守操作
			{4,5}		**データバリア操作**
			6		CDSR、キャッシュダーティステータス レジスタ
		c11	{0-2}		キャッシュ保守操作
		c12	{4,5}	‡	**ブロック転送サポート操作(オプション)**
		c13	1		キャッシュプリフェッチ操作(オプション)
		c14	{0-3}	‡	キャッシュ保守操作
c8	0	{c5-c7}	{0-2}		TLB保守操作
c9	0	{c0,c1}	{0,1}		キャッシュロックダウン(フォーマットC)およびTCM領域レジスタ
		*c1	*{2,3}		† TCM非セキュアアクセス制御レジスタ
		c2	0		TCMSR、TCM選択レジスタ
		{c5,c6}	{0,1}		キャッシュロックダウン(フォーマットD)操作
		*c8	*0		† CBOR、キャッシュ動作オーバライドレジスタ
c10	0	c0	{0,1}		TLBロックダウンレジスタ
		*c2	*{0,1}		* TEX再マップレジスタ(PRRR、NMRR)
		{c4,c8}	{0,1}	‡	TLBロックダウン操作およびレジスタ(オプション)
c11	{0-7}	{c0-c8,c15}	{0-7}	‡	TCM操作のDMAサポート用に予約
c12	0	*c0	*{0,1}		† セキュリティ拡張機能レジスタ
		*c1	*0		† ISR、割り込みステータスレジスタ
c13	0	c0	0		FCSEIDR、FCSE PIDレジスタ
			1		CONTEXTIDR、コンテキストIDレジスタ
			*{2-4}		*ソフトウェアスレッドIDレジスタ
c15	{0-7}	{c0-c15}	{0-7}	‡	実装定義レジスタ

読み出し専用 | 読み出し/書き込み | 書き込み専用 | **太字のテキスト = ユーザモードでアクセス可能**

* ARMv6のすべてのバージョンで使用不可 †セキュリティ拡張機能が実装されている場合 ‡ アクセスは動作によって異なります

図 G-1 ARMv6 VMSA 実装の CP15 レジスタ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.7.2 Organization of CP15 registers for an ARMv6 PMSA implementation

Figure G-2 shows the CP15 registers in an ARMv6 PMSA implementation:

Figure G-2 CP15 registers in an ARMv6 PMSA implementation

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.7.2 ARMv6 PMSA 実装での CP15 レジスタの編成

ARMv6 PMSA 実装の CP15 レジスタを、図 G-2 に示します。

図 G-2 ARMv6 PMSA 実装の CP15 レジスタ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.7.3 c0, ID support

ARMv6 implementations include a Main ID Register, see *c0, Main ID Register (MIDR)* on page B3-81. In this register, the architecture variant field either takes the assigned ARMv6 value or indicates support for an identification scheme based on a set of CPUID registers. The CPUID identification scheme is required in ARMv7 and recommended for ARMv6, and is described in Chapter B5 *The CPUID Identification Scheme*.

Three other ID registers provide information about cache, TCM, and TLB provisions. From ARMv6K, there is also a Multiprocessor Affinity Register.

All of the CP15 c0 ID registers are read-only registers, They are accessed using `MRC` instructions, as shown in Table G-5.

Table G-5 ID register support

Register	CRn	opc1	CRm	opc2
MIDR, Main ID Register	c0	0	c0	0
CTR, Cache Type ID Register	c0	0	c0	1
TCMTR, TCM Type Register	c0	0	c0	2
TLBTR, TLB Type Register[a]	c0	0	c0	3
MPUIR, MPU Type Register[c]	c0	0	c0	4
MPIDR, Multiprocessor Affinity Register[b]	c0	0	c0	5
Aliases of MIDR	c0	0	c0	3[c], 4[a], 5[d], 6, 7
CPUID registers, if implemented	c0	0	c1	0-7
	c0	0	c2	0-5

a. VMSA processors only.
b. ARMv6K processors with VMSA only.
c. PMSA processors only.
d. All ARMv6 processors except ARMv6K VMSA implementations.

The Cache Type Register is as defined for ARMv4 and ARMv5, see *c0, Cache Type Register (CTR)* on page AppxH-35. In ARMv6, the CType values of 0b0110, and 0b0111 are reserved and must not be used.

Note

The ARMv6 format of the Cache Type Register is significantly different from the ARMv7 implementation described in *c0, Cache Type Register (CTR)* on page B3-83. However, the general properties described by the register, and the access rights for the register, are unchanged.

The TCM Type Register is defined in *c0, TCM Type Register (TCMTR)* on page AppxG-33.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.7.3 c0、ID のサポート

ARMv6 実装には、メイン ID レジスタが含まれています。*c0、メイン ID レジスタ (MIDR)* P. B3-81 を参照して下さい。このレジスタのアーキテクチャバリアント フィールドは、割り当て済みの ARMv6 値を取得するか、または CPUID レジスタのセットに基づく識別方式のサポートを示すかのいずれかです。CPUID 識別方式は ARMv7 では必須で、ARMv6 では推奨されます。これについては、第 B5 章*「CPUID 識別方式」*で説明されています。

他の 3 つの ID レジスタでは、キャッシュ、TCM、TLB 機構に関する情報が提供されます。ARMv6K およびそれ以降には、マルチプロセッサ類似性レジスタも存在します。

CP15 c0 ID レジスタはすべて読み出し専用レジスタです。表 G-5 に示されているように、これらのレジスタには `MRC` 命令を使用してアクセスします。

表 G-5 ID レジスタのサポート

レジスタ	CRn	opc1	CRm	opc2
MIDR、メイン ID レジスタ	c0	0	c0	0
CTR、キャッシュタイプ ID レジスタ	c0	0	c0	1
TCMTR、TCM タイプレジスタ	c0	0	c0	2
TLBTR、TLB タイプレジスタ[a]	c0	0	c0	3
MPUIR、MPU タイプレジスタ[c]	c0	0	c0	4
MPIDR、マルチプロセッサ類似性レジスタ[b]	c0	0	c0	5
MIDR のエイリアス	c0	0	c0	3[c]、4[a]、5[d]、6、7
CPUID レジスタ（実装されている場合）	c0	0	c1	0 ～ 7
	c0	0	c2	0 ～ 5

a. VMSA プロセッサのみ
b. VMSA を搭載した ARMv6K プロセッサのみ
c. PMSA プロセッサのみ
d. ARMv6K VMSA 実装以外のすべての ARMv6 プロセッサ

キャッシュタイプ レジスタは、ARMv4 および ARMv5 での定義と同様です。*c0、キャッシュタイプ レジスタ (CTR)* P. 付録 H-35 を参照して下さい。ARMv6 では、CType の値 0b0110 および 0b0111 は予約されており、使用できません。

注

キャッシュタイプ レジスタの ARMv6 でのフォーマットは、*c0、キャッシュタイプ レジスタ (CTR)* P. B3-83 で説明されている ARMv7 実装の場合とは大きく異なります。ただし、レジスタによって記述される汎用プロパティ、およびレジスタのアクセス権限は変更されていません。

TCM タイプレジスタは、*c0、TCM タイプレジスタ (TCMTR)* P. 付録 G-33 で定義されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The TLB Type ID Register and the Multiprocessor Affinity Register are as defined for ARMv7, see:

- *c0, TLB Type ID Register (TLBTR)*
- *c0, Multiprocessor Affinity Register (MPIDR)* on page B3-87.

The MPU Type Register is as defined for ARMv7, see *c0, MPU Type Register (MPUIR)* on page B4-36. In an ARMv6 PMSA implementation, if the MPU is not implemented use of the default memory map is optional.

c0, TCM Type Register (TCMTR)

The TCMTR must be implemented in ARMv6 and ARMv7. In ARMv7, the register can have a different format from that given here, see *c0, TCM Type Register (TCMTR)* on page B3-85.

In ARMv7, TCM support is IMPLEMENTATION DEFINED. For ARMv6, see *c9, TCM support* on page AppxG-46 and *c9, TCM Non-Secure Access Control Registers, DTCM-NSACR and ITCM-NSACR* on page AppxG-51 where the Security Extensions are supported.

31 29	28 19	18 16	15 3	2 0
0 0 0	Reserved	DTCM	Reserved	ITCM

Bits [31:29] Set to 0b000 before ARMv7.

Bits [28:19,15:3]

Reserved.

DTCM, Bits [18:16] Indicate the number of Data TCMs implemented. This value lies in the range 0 to 4, 0b000 to 0b100. All other values are reserved.

ITCM, Bits [2:0] Indicate the number of Instruction or Unified TCMs implemented. This value lies in the range 0 to 4, 0b000 to 0b100. All other values are reserved.

Instruction TCMs are accessible to both instruction and data sides.

c0, TLB Type ID Register (TLBTR)

In an ARMv6 VMSA implementation the TLB Type Register, TLBTR, is a read-only register that defines whether the implementation provides separate instruction and data TLBs, or a unified TLB. It also defines the number of lockable TLB entries. The ARMv7-A description of the register describes the general features of the register and how to access it. See *c0, TLB Type Register (TLBTR)* on page B3-86. However, the register format is different in ARMv6. The ARMv6 format of the TLBTR is:

31 24	23 16	15 8	7 1	0
Reserved	I_nlock	D_nlock	Reserved	nU

Bits [31:24, 7:1] Reserved, UNK.

I_nlock, bits [23:16] Number of lockable entries in the instruction TLB. The value of this field gives the number of lockable entries, between 0b00000000 for no lockable entries, and 0b11111111 for 255 lockable entries.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

TLB タイプ ID レジスタおよびマルチプロセッサ類似性レジスタは、ARMv7 での定義と同様です。次のセクションを参照して下さい。

- *c0、TLB タイプ ID レジスタ (TLBTR)*
- *c0、マルチプロセッサ類似性レジスタ (MPIDR)* P. B3-87

MPU タイプレジスタは、ARMv7 での定義と同様です。*c0、MPU タイプレジスタ (MPUIR)* P. B4-36 を参照して下さい。ARMv6 PMSA 実装では、MPU が実装されていない場合、デフォルトのメモリマップの使用はオプションです。

c0、TCM タイプレジスタ (TCMTR)

ARMv6 および ARMv7 では、TCMTR を実装する必要があります。ARMv7 では、レジスタはここで示されているフォーマットとは別のフォーマットにすることができます。*c0、TCM タイプレジスタ (TCMTR)* P. B3-85 を参照して下さい。

ARMv7 では、TCM のサポートは実装定義です。ARMv6 の場合は、*c9、TCM のサポート* P. 付録 G-46 および *c9、TCM 非セキュアアクセス制御レジスタ、DTCM-NSACR および ITCM-NSACR* P. 付録 G-51 で、セキュリティ拡張機能がサポートされている場合を参照して下さい。

31 29	28 19	18 16	15 3	2 0
0 0 0	予約	DTCM	予約	ITCM

ビット [31:29] ARMv7 以前では、0b000 に設定されます。

ビット [28:19、15:3] 予約

DTCM、ビット [18:16] 実装されているデータ TCM の数を示します。この値の範囲は 0 ～ 4(0b000 ～ 0b100) です。他の値はすべて予約されています。

ITCM、ビット [2:0] 実装されている命令 TCM または統一 TCM の数を示します。この値の範囲は 0 ～ 4(0b000 ～ 0b100) です。他の値はすべて予約されています。

命令 TCM は、命令側およびデータ側の両方からアクセス可能です。

c0、TLB タイプ ID レジスタ (TLBTR)

ARMv6 VMSA 実装では、TLB タイプレジスタ (TLBTR) は読み出し専用レジスタです。このレジスタにより、実装で個別の命令 TLB およびデータ TLB、または統一 TLB のどちらが提供されるかが定義されます。また、ロック可能な TLB エントリの数も定義されます。ARMv7-A のレジスタの説明には、レジスタの一般的な機能およびレジスタへのアクセス方法が示されています。*c0、TLB タイプレジスタ (TLBTR)* P. B3-86 を参照して下さい。ただし、レジスタのフォーマットは ARMv6 の場合とは異なります。ARMv6 での TLBTR のフォーマットは次のとおりです。

31 24	23 16	15 8	7 1	0
予約	I_nlock	D_nlock	予約	nU

ビット [31:24、7:1] 予約、UNK

I_nlock、ビット [23:16] 命令 TLB のロック可能なエントリの数。このフィールドの値はロック可能なエントリの数を示し、0b00000000（ロック可能なエントリがない場合）と 0b11111111（ロック可能なエントリの数が 255 の場合）の間の値です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

When nU == 0 this field is reserved, UNK.

D_nlock, bits [15:8] Number of lockable entries in the data TLB. The value of this field gives the number of lockable entries, between 0b00000000 for no lockable entries, and 0b11111111 for 255 lockable entries.

nU, bit [0] Not Unified TLB. Indicates whether the implementation has a unified TLB:

nU == 0 Unified TLB.

nU == 1 Separate instruction and data TLBs.

G.7.4 c1, System control support

ARMv6 implements the same system control registers as ARMv7:

- for a VMSA implementation, see *CP15 c1, System control registers* on page B3-96
- for a PMSA implementation, see *CP15 c1, System control registers* on page B4-44.

c1, System Control Register (SCTLR)

This register is the primary system configuration register in CP15. It is defined differently for VMSA and PMSA.

In a VMSAv6 implementation, the format of the SCTLR is:

In an ARMv6K VMSA implementation, the format of the SCTLR is:

Note

Where the Security Extensions are implemented, some SCTLR bits are banked as described in *c1, System Control Register (SCTLR)* on page B3-96.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

nU = 0 の場合、このフィールドは予約、UNK です。

D_nlock、ビット [15:8] データ TLB のロック可能なエントリの数。このフィールドの値はロック可能なエントリの数を示し、0b00000000（ロック可能なエントリがない場合）と 0b11111111（ロック可能なエントリの数が 255 の場合）の間の値です。

nU、ビット [0] 統一 TLB の非実装。実装に統一 TLB が含まれているかどうかを示します。

nU = 0 統一 TLB

nU = 1 独立した命令 TLB とデータ TLB

G.7.4 c1、システム制御のサポート

ARMv6 では、ARMv7 の場合と同じシステム制御レジスタが実装されます。

- VMSA 実装の場合、*CP15 c1、システム制御レジスタ* P. B3-96 を参照して下さい。
- PMSA 実装の場合、*CP15 c1、システム制御レジスタ* P. B4-44 を参照して下さい。

c1、システム制御レジスタ (SCTLR)

このレジスタは、CP15 の 1 次システム構成レジスタです。このレジスタの定義は、VMSA と PMSA では異なります。

VMSAv6 実装では、SCTLR のフォーマットは次のとおりです。

ARMv6K VMSA 実装では、SCTLR のフォーマットは次のとおりです。

注

セキュリティ拡張機能が実装されている場合、SCTLR の一部のビットは *c1、システム制御レジスタ (SCTLR)* P. B3-96 で説明されているようにバンクされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In a PMSAv6 implementation, the format of the SCTLR is:

The differences from ARMv7 are:

- ARMv6 does not support the SCTLR.IE and SCTLR.BR, bits 31 and 17.
- ARMv7 does not support:
  - — The L2, L4, R, S, B, and W bits. These bits provide legacy support with ARMv4 and ARMv5. See *c1, System Control Register (SCTLR)* on page AppxH-39 for their definition.

    The B bit must also meet the requirements defined in *Endian support* on page AppxG-7.
  - — The U bit. This bit is always 1 in ARMv7. It selects the ARMv4 and ARMv5 or the ARMv6 and ARMv7 alignment model. For details see *Alignment* on page AppxG-6.
  - — The XP bit. This bit is always 1 in ARMv7. The bit selects the virtual memory support model of ARMv6 and ARMv7 when SCTLR.XP = 1, and the legacy support for ARMv4 and ARMv5 when SCTLRR.XP = 0. For ARMv6 and ARMv7 support, see *VMSAv6 translation table format* on page AppxG-26 and Chapter B3 *Virtual Memory System Architecture (VMSA)*, and for ARMv4 and ARMv5 support, see *Legacy translation table format* on page AppxG-25 and *Virtual memory support* on page AppxH-21.
- The TE bit is defined for ARMv6T2 only. In ARMv6T2 it is the same as in ARMv7.

For the definition of bits supported in ARMv6 and ARMv7, see:

- *c1, System Control Register (SCTLR)* on page B3-96 for a VMSA implementation
- *c1, System Control Register (SCTLR)* on page B4-45 for a PMSA implementation.

G.7.5 c1, VMSA Security Extensions support

An ARMv6 implementation that includes the Security Extensions provides:

- the banking of bits in SCTLR, see *c1, System Control Register (SCTLR)* on page AppxG-34
- features that are include in an ARMv7 implementation of the Security Extensions, see:
  - — *c1, Secure Configuration Register (SCR)* on page B3-106
  - — *c1, Secure Debug Enable Register (SDER)* on page B3-108
  - — *c1, Non-Secure Access Control Register (NSACR)* on page B3-110

In addition, ARMv6 defines the following additional bits in the NSACR:

Bit [18], DMA DMA control register access for support in CP15 c11 in the Non-secure address space. For more information, see *c11, DMA support* on page AppxG-54.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

PMSAv6 実装では、SCTLR のフォーマットは次のとおりです。

ARMv7 との相違点は次のとおりです。

- ARMv6 では、SCTLR.IE および SCTLR.BR（ビット 31 と 17）はサポートされません。
- ARMv7 では、次のビットはサポートされません。
  - – L2、L4、R、S、B、W ビット。これらのビットは、ARMv4 および ARMv5 で過去のサポートを提供します。これらのビットの定義については *c1、システム制御レジスタ (SCTLR)* P. 付録 H-39 を参照して下さい。

    B ビットは、*エンディアンのサポート* P. 付録 G-7 で定義されている要件も満たす必要があります。
  - – U ビット。ARMv7 では、このビットは常に 1 です。このビットにより、ARMv4 および ARMv5、または ARMv6 および ARMv7 アライメントモデルが選択されます。詳細については、*アライメント* P. 付録 G-6 を参照して下さい。
  - – XP ビット。ARMv7 では、このビットは常に 1 です。このビットにより、SCTLR.XP = 1 の場合は ARMv6 および ARMv7 の仮想メモリ サポートモデルが選択され、SCTLRR.XP = 0 の場合は ARMv4 および ARMv5 の過去のサポートが選択されます。ARMv6 および ARMv7 のサポートについては、*VMSAv6 の変換テーブルのフォーマット* P. 付録 G-26 および第 B3 章 *「仮想メモリシステム アーキテクチャ (VMSA)」*を、ARMv4 および ARMv5 のサポートについては、*過去の変換テーブルフォーマット* P. 付録 G-25 および*仮想メモリのサポート* P. 付録 H-21 を参照して下さい。
- TE ビットは、ARMv6T2 でのみ定義されます。ARMv6T2 では、このビットは ARMv7 の場合と同じです。

ARMv6 および ARMv7 でサポートされるビットの定義については、次のセクションを参照して下さい。

- *c1、システム制御レジスタ (SCTLR)* P. B3-96（VMSA 実装の場合）
- *c1、システム制御レジスタ (SCTLR)* P. B4-45 （PMSA 実装の場合）

G.7.5 c1、VMSA セキュリティ拡張機能のサポート

セキュリティ拡張機能を含む ARMv6 実装では、次の機能が提供されます。

- SCTLR のビットのバンク処理（*c1、システム制御レジスタ (SCTLR)* P. 付録 G-34 を参照）
- セキュリティ拡張機能の ARMv7 実装に含まれる機能（次のセクションを参照）
  - – *c1、セキュア構成レジスタ (SCR)* P. B3-106
  - – *c1、セキュアデバッグ イネーブルレジスタ (SDER)* P. B3-108
  - – *c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3-110

さらに、ARMv6 では NSACR に次の追加ビットが定義されます。

ビット [18]、DMA

非セキュアなアドレス空間における CP15 c11 のサポートでの DMA 制御レジスタアクセス。詳細については、*c11、DMA のサポート* P. 付録 G-54 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Bit [17], TL TLB Lockdown Register access for support in CP15 c10 in the Non-secure address space. For information on TLB lockdown support in ARMv6 see *c10, VMSA TLB lockdown support* on page AppxG-53.

Bit [16], CL Cache Lockdown Register access for support in CP15 c9 in the Non-secure address space. For information on cache lockdown support in ARMv6 see *c9, Cache lockdown support* on page AppxG-45.

In all cases:

- a value of 0 in the bit position specifies that the associated registers cannot be accessed in the Non-secure address space
- a value of 1 in the bit position specifies that the associated registers can be accessed in the Secure and Non-secure address spaces.

Support of these additional bits and more details on how DMA support for TCMs, TLB lockdown, and cache lockdown are inhibited in the Non-secure address space is IMPLEMENTATION DEFINED.

G.7.6 c2 and c3, VMSA memory protection and control registers

ARMv6 and ARMv7 provide the same CP15 support:

- two Translation Table Base Registers, TTBR0 and TTBR1
- a Translation Table Base Control Register, TTBCR
- a Domain Access Control Register, DACR.

The translation table registers are defined in *CP15 c2, Translation table support registers* on page B3-113.

The *Domain Access Control Register* (DACR) is as defined in *c3, Domain Access Control Register (DACR)* on page B3-119.

Note

When the Security Extensions are implemented, these registers are Banked registers.

G.7.7 c5 and c6, VMSA memory system support

The support in ARMv6 is the same as ARMv7 with the following exceptions:

- Bit 12 of the data and instruction fault status registers is not defined in ARMv6. See *c5, Data Fault Status Register (DFSR)* on page B3-121 and *c5, Instruction Fault Status Register (IFSR)* on page B3-122.
- The *Auxiliary Data Fault Status Register* (ADFSR) and the *Auxiliary Instruction Fault Status Register* (AIFSR) are not defined in ARMv6. See *c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B3-123.
- The Access Flag faults shown in Table B3-11 on page B3-50 and Table B3-12 on page B3-51 are only supported in ARMv6K.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ビット [17]、TL

非セキュアなアドレス空間における CP15 c10 のサポートでの TLB ロックダウン レジスタ アクセス。ARMv6 における TLB ロックダウンのサポートの詳細については、*c10、VMSA TLB ロックダウンのサポート* P. 付録 G-53 を参照して下さい。

ビット [16]、CL

非セキュアなアドレス空間における CP15 c9 のサポートでのキャッシュロックダウン レジスタ アクセス。ARMv6 におけるキャッシュロックダウンのサポートの詳細については、*c9、キャッシュロックダウンのサポート* P. 付録 G-45 を参照して下さい。

すべての場合について、次の規則が適用されます。

- ビット位置の値が 0 の場合、非セキュアなアドレス空間では関連付けられたレジスタにアクセスできないことを示します。
- ビット位置の値が 1 の場合、セキュアおよび非セキュアなアドレス空間で、関連付けられたレジスタにアクセスできることを示します。

これらの追加ビットのサポート、および非セキュアなアドレス空間において TCM、TLB ロックダウン、キャッシュロックダウンの DMA サポートを禁止する方法の詳細は実装定義です。

G.7.6 c2 および c3、VMSA メモリ保護および制御レジスタ

ARMv6 および ARMv7 では、同じ CP15 サポートが提供されます。

- 2 つの変換テーブル ベースレジスタ、TTBR0 および TTBR1
- 変換テーブルベース制御レジスタ、TTBCR
- ドメインアクセス制御レジスタ、DACR

変換テーブルレジスタは *CP15 c2、変換テーブルサポート レジスタ* P. B3-113 で定義されています。

ドメインアクセス制御レジスタ (DACR) は、*c3、ドメインアクセス制御レジスタ (DACR)* P. B3-119 で定義されているものと同じです。

注

セキュリティ拡張機能が実装されている場合、これらのレジスタはバンクレジスタです。

G.7.7 c5 および c6、VMSA メモリシステムのサポート

次の例外を除いて、ARMv6 でのサポートは ARMv7 と同じです。

- ARMv6 では、データフォールトステータス レジスタおよび命令フォールトステータス レジスタのビット 12 は定義されていません。*c5、データフォールト ステータスレジスタ (DFSR)* P. B3-121 および *c5、命令フォールト ステータスレジスタ (IFSR)* P. B3-122 を参照して下さい。
- ARMv6 では、*補助データフォールトステータス レジスタ* (ADFSR) および *補助命令フォールトステータス レジスタ* (AIFSR) は定義されていません。*c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B3-123 を参照して下さい。
- P. B3-50 の表 B3-11 および P. B3-51 の表 B3-12 に示されているアクセスフラグ フォールトは、ARMv6K でのみサポートされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

- Before ARMv7, the DFAR was called the *Fault Address Register* (FAR).
- If the Security Extensions are implemented, these registers are Banked registers.
- In ARMv6 variants other than ARMv6T2, the IFAR is optional.

G.7.8 c5 and c6, PMSA memory system support

The support in ARMv6 is the same as ARMv7 with the following exceptions:

- The SCTLR.BR bit, bit [17], is not supported in ARMv6, see *c1, System Control Register (SCTLR)* on page B4-45.
- Bit 12 of the data and instruction fault status registers is not defined in ARMv6. See *c5, Data Fault Status Register (DFSR)* on page B4-55 and *c5, Instruction Fault Status Register (IFSR)* on page B4-56.
- The ADFSR and the AIFSR are not defined in ARMv6. See *c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B4-56.
- Subregions are not supported. This means that DRSR[15:8] and IRSR[15:8] are not defined in ARMv6. See *c6, Data Region Size and Enable Register (DRSR)* on page B4-62 and *c6, Instruction Region Size and Enable Register (IRSR)* on page B4-63.

Note

- Before ARMv7, the DFAR was called the *Fault Address Register* (FAR).
- In ARMv6 variants other than ARMv6T2, the IFAR is optional.

G.7.9 c6, Watchpoint Fault Address Register (DBGWFAR)

From v6.1 of the Debug architecture, this register is also implemented as DBGWFAR in CP14, and the use of CP15 DBGWFAR is deprecated.

In an ARMv6 implementation that includes the Security Extensions, CP15 DBGWFAR is a Secure register, and can be accessed only from Secure privileged modes. For more information, see *Restricted access CP15 registers* on page B3-73.

For more information about this register see *Effects of debug exceptions on CP15 registers and the DBGWFAR* on page C4-4 and *Effect of entering Debug state on CP15 registers and the DBGWFAR* on page C5-4.

Table G-6 Debug fault address support

Register	CRn	opc1	CRm	opc2
Watchpoint Fault Address Register, DBGWFAR	c6	0	c0	1

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

注

- ARMv7 以前では、DFAR は*フォールト アドレス レジスタ*(FAR) と呼ばれます。
- セキュリティ拡張機能が実装されている場合、これらのレジスタはバンクレジスタです。
- ARMv6T2 以外の ARMv6 バリアントでは、IFAR はオプションです。

G.7.8 c5 および c6、PMSA メモリシステムのサポート

次の例外を除いて、ARMv6 でのサポートは ARMv7 と同じです。

- ARMv6 では、SCTLR.BR ビット（ビット [17]）はサポートされません。*c1、システム制御レジスタ (SCTLR)* P. B4-45 を参照して下さい。
- ARMv6 では、データフォールトステータス レジスタおよび命令フォールトステータス レジスタのビット 12 は定義されていません。*c5、データフォールト ステータスレジスタ (DFSR)* P. B4-55 および *c5、命令フォールト ステータスレジスタ (IFSR)* P. B4-56 を参照して下さい。
- ARMv6 では、ADFSR および AIFSR は定義されていません。*c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B4-56 を参照して下さい。
- サブ領域はサポートされていません。これは、ARMv6 では DRSR[15:8] および IRSR[15:8] が定義されていないことを意味します。*c6、データ領域サイズおよびイネーブルレジスタ (DRSR)* P. B4-62 および *c6、命令領域サイズおよびイネーブルレジスタ (IRSR)* P. B4-63 を参照して下さい。

注

- ARMv7 以前では、DFAR は*フォールト アドレス レジスタ*(FAR) と呼ばれます。
- ARMv6T2 以外の ARMv6 バリアントでは、IFAR はオプションです。

G.7.9 c6、ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)

デバッグアーキテクチャの v6.1 およびそれ以降では、このレジスタは CP14 の DBGWFAR としても実装されており、CP15 DBGWFAR の使用は非推奨です。

セキュリティ拡張機能を含む ARMv6 実装では、CP15 DBGWFAR はセキュアレジスタで、セキュア特権モードからのみアクセスできます。詳細については、*限定アクセス CP15 レジスタ* P. B3-73 を参照して下さい。

このレジスタの詳細については、*CP15 レジスタおよび DBGWFAR へのデバッグ例外の影響* P. C4-4 および*デバッグ状態への遷移が CP15 レジスタおよび DBGWFAR に及ぼす影響* P. C5-4 を参照して下さい。

表 G-6 デバッグ フォールトアドレスのサポート

レジスタ	CRn	opc1	CRm	opc2
ウォッチポイントフォールトアドレス レジスタ、DBGWFAR	c6	0	c0	1

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.7.10 c7, Cache operations

Table G-7 shows the cache operations defined for ARMv6. They are performed as MCR instructions and only operate on a level 1 cache associated with a specific processor. The equivalent operations in ARMv7 operate on multiple levels of cache. See *CP15 c7, Cache maintenance and other functions* on page B3-126. For a list of required operations in ARMv6, see *Cache support* on page AppxG-21. Support of additional operations is IMPLEMENTATION DEFINED.

Table G-7 Cache operation support

Operation	CRn	opc1	CRm	opc2
Invalidate instruction cache[a]	c7	0	c5	0
Invalidate instruction cache line by MVA[a]	c7	0	c5	1
Invalidate instruction cache line by set/way	c7	0	c5	2
Flush entire branch predictor array[a]	c7	0	c5	6
Flush branch predictor array entry by MVA [a]	c7	0	c5	7
Invalidate data cache	c7	0	c6	0
Invalidate data cache line by MVA[a]	c7	0	c6	1
Invalidate data cache line by set/way[a]	c7	0	c6	2
Invalidate unified cache, or instruction cache and data cache	c7	0	c7	0
Invalidate unified cache line by MVA	c7	0	c7	1
Invalidate unified cache line by set/way	c7	0	c7	2
Clean data cache	c7	0	c10	0
Clean data cache line by MVA[a]	c7	0	c10	1
Clean data cache line by set/way[a]	c7	0	c10	2
Test and Clean data cache[b]	c7	0	c10	3
Cache Dirty Status Register[c]	c7	0	c10	6
Clean entire unified cache	c7	0	c11	0
Clean unified cache line by MVA[a]	c7	0	c11	1
Clean unified cache line by set/way	c7	0	c11	2
Prefetch instruction cache line by MVA[d]	c7	0	c13	1

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.7.10 c7、キャッシュ操作

ARMv6 で定義されているキャッシュ操作を、表 G-7 に示します。これらの操作は MCR 命令として実行され、特定のプロセッサに関連付けられたレベル 1 キャッシュでのみ動作します。等価な ARMv7 の操作は、キャッシュの複数のレベルで動作します。*CP15 c7、キャッシュ保守機能とその他の機能* P. B3-126 を参照して下さい。ARMv6 で必須の操作のリストについては、*キャッシュのサポート* P. 付録 G-21 を参照して下さい。追加操作のサポートは実装定義です。

表 G-7 キャッシュ操作のサポート

動作	CRn	opc1	CRm	opc2
命令キャッシュの無効化[a]	c7	0	c5	0
MVA による命令キャッシュラインの無効化[a]	c7	0	c5	1
セット / ウェイによる命令キャッシュラインの無効化	c7	0	c5	2
分岐予測器アレイ全体のフラッシュ[a]	c7	0	c5	6
MVA による分岐予測器アレイエントリのフラッシュ[a]	c7	0	c5	7
データキャッシュの無効化	c7	0	c6	0
MVA によるデータ キャッシュラインの無効化[a]	c7	0	c6	1
セット / ウェイによるデータ キャッシュラインの無効化[a]	c7	0	c6	2
統一キャッシュ、または命令キャッシュとデータキャッシュの無効化	c7	0	c7	0
MVA による統一キャッシュラインの無効化	c7	0	c7	1
セット / ウェイによる統一キャッシュラインの無効化	c7	0	c7	2
データキャッシュのクリーニング	c7	0	c10	0
MVA によるデータ キャッシュラインのクリーニング[a]	c7	0	c10	1
セット / ウェイによるデータ キャッシュラインのクリーニング[a]	c7	0	c10	2
データキャッシュのテストとクリーニング[b]	c7	0	c10	3
キャッシュダーティステータス レジスタ[c]	c7	0	c10	6
統一キャッシュ全体のクリーニング	c7	0	c11	0
MVA による統一キャッシュラインのクリーニング[a]	c7	0	c11	1
セット / ウェイによる統一キャッシュラインのクリーニング	c7	0	c11	2
MVA による命令キャッシュラインのプリフェッチ[d]	c7	0	c13	1

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table G-7 Cache operation support (continued)

Operation	CRn	opc1	CRm	opc2
Clean and Invalidate data cache	c7	0	c14	0
Clean and Invalidate data cache line by MVA[a]	c7	0	c14	1
Clean and Invalidate data cache line by set/way[a]	c7	0	c14	2
Test and Clean and Invalidate data cache[b]	c7	0	c14	3
Clean and Invalidate unified cache line by MVA	c7	0	c15	1
Clean and Invalidate unified cache line by set/way	c7	0	c15	2

a. These are the only cache operations available in ARMv7. The corresponding ARMv7 operations are multi-level operations, and the data cache operations are defined as data or unified cache operations.
b. For more information about these cache operations see *Test and clean operations* on page AppxH-50.
c. Used with the Clean or Clean and Invalidate entire data cache and entire unified cache operations.
d. VMSA implementations only, used with TLB lockdown. See *The TLB lock by entry model* on page AppxH-60.

c7, Cache Dirty Status Register (CDSR)

The Cache Dirty Status Register, CDSR, indicates whether the data or unified cache has been written to since the last successful cache clean. For more information, see *Cleaning and invalidating operations for the entire data cache* on page AppxG-40.

The Cache Dirty Status Register is:

- a 32-bit read-only register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Banked register.

The format of the Cache Dirty Status Register is:

31 … 1	0
Reserved	C

Bits [31:1] Reserved, UNK/SBZP.

C, bit [0] Cache Dirty Status. The meaning of this bit is:

0 Cache clean. No write has hit the cache since the last cache clean or reset successfully cleaned the cache.

1 The cache might contain dirty data.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 G-7 キャッシュ操作のサポート （続き）

動作	CRn	opc1	CRm	opc2
データキャッシュのクリーニングと無効化	c7	0	c14	0
MVA によるデータ キャッシュラインのクリーニングと無効化[a]	c7	0	c14	1
セット / ウェイによるデータ キャッシュラインのクリーニングと無効化[a]	c7	0	c14	2
データキャッシュのテスト、クリーニング、および無効化[b]	c7	0	c14	3
MVA による統一キャッシュラインのクリーニングと無効化	c7	0	c15	1
セット / ウェイによる統一キャッシュラインのクリーニングと無効化	c7	0	c15	2

a. ARMv7 で使用可能なキャッシュ操作はこれらだけです。対応する ARMv7 操作は複数レベルの操作で、データキャッシュ操作はデータまたは統一キャッシュの操作として定義されます。
b. これらのキャッシュ操作の詳細については、*テスト操作とクリーニング操作* P. 付録 H-50 を参照して下さい。
c. データキャッシュ全体および統一キャッシュ全体のクリーニング、またはクリーニングと無効化操作で使用されます。
d. VMSA 実装のみ。TLB ロックダウンとともに使用されます。*エントリモデルによる TLB ロック* P. 付録 H-60 を参照して下さい。

c7、キャッシュダーティステータス レジスタ (CDSR)

キャッシュダーティステータス レジスタ (CDSR) は、最後にキャッシュのクリーニングが正常に行われてから、データキャッシュまたは統一キャッシュに書き込みが行われたかどうかを示します。詳細については、*データキャッシュ全体のクリーニングと無効化の操作* P. 付録 G-40 を参照して下さい。

キャッシュダーティステータス レジスタには、次の特徴があります。

- 32 ビットの読み出し専用レジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、バンクレジスタです。

キャッシュダーティステータス レジスタのフォーマットは次のとおりです。

31 … 1	0
予約	C

ビット [31:1] 予約、UNK/SBZP

C（ビット [0]）

キャッシュダーティ ステータス。このビットの意味は次のとおりです。

0 キャッシュはクリーンです。最後にキャッシュのクリーニングまたはリセットにより、正常にキャッシュがクリーニングされた後で、キャッシュに書き込みがヒットしていません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Accessing the Cache Dirty Status Register

To access the Cache Dirty Status Register you read the CP15 registers with <opc1> set to 0, <CRn> set to c7, <CRm> set to c10, and <opc2> set to 6. For example:

```
MRC p15, 0, <Rt>, c7, c10, 6
```

Cleaning and invalidating operations for the entire data cache

The CP15 c7 encodings include operations for cleaning the entire data cache, and for performing a clean and invalidate of the entire data cache. If these operations are interrupted, the LR value that is captured on the interrupt is (address of instruction that launched the cache operation + 4). This permits the standard return mechanism for interrupts to restart the operation.

If a particular operation requires that the cache is clean, or clean and invalid, then it is essential that the sequence of instructions for cleaning or cleaning and invalidating the cache can cope with the arrival of an interrupt at any time when interrupts are not disabled. This is because interrupts might write to a previously cleaned cache block. For this reason, the Cache Dirty Status Register indicates whether the cache has been written to since the last successful cache clean.

You can interrogate the Cache Dirty Status Register to determine whether the cache is clean, and if you do this while interrupts are disabled, a subsequent operation can rely on having a clean cache. The following sequence illustrates this approach.

```
; The following code assumes interrupts are enabled at this point.
Loop1
        MOV     R1, #0
        MCR     p15, 0, R1, c7, c10, 0          ; Clean data cache. For Clean and Invalidate,
                                                ;  use MCR p15, 0, R1, c7, c14, 0 instead
        MRS     R2, CPSR                        ; Save PSR context
        CPSID   iaf                             ; Disable interrupts
        MRC     p15, 0, R1, c7, c10, 6          ; Read Cache Dirty Status Register
        TST     R1, #1                          ; Check if it is clean
        BEQ     UseClean
        MSR     CPSR_xc, R2                     ; Re-enable interrupts
        B       Loop1                           ; Clean the cache again
UseClean
        Do_Clean_Operations                     ; Perform whatever operation relies on
                                                ;  the cache being clean or clean & invalid.
                                                ;  To reduce impact on interrupt latency,
                                                ;  this sequence should be short.
        MCR     p15, 0, R1, c7, c6, 0           ; Optional. Can use this Invalidate all command
                                                ;  to invalidate a Clean loop.
        MSR     CPSR_xc, R2                     ; Re-enable interrupts
```

Note

The long cache clean operation is performed with interrupts enabled throughout this routine.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

1 キャッシュにダーティデータが含まれている可能性があります。

キャッシュダーティステータス レジスタへのアクセス

キャッシュダーティステータス レジスタにアクセスするには、<opc1> を 0、<CRn> を c7、<CRm> を c10、<opc2> を 6 に設定して、CP15 レジスタを読み出します。次に例を示します。

```
MRC p15, 0, <Rt>, c7, c10, 6
```

データキャッシュ全体のクリーニングと無効化の操作

CP15 c7 エンコードには、データキャッシュ全体のクリーニング、およびデータキャッシュ全体のクリーニングと無効化の操作が含まれています。これらの操作で割り込みが発生した場合、割り込み時に取り込まれる LR 値は（キャッシュ操作を開始した命令のアドレス + 4）です。これにより、割り込みの標準的な復帰機構を使用して操作を再開できます。

特定の操作において、キャッシュがクリーン、またはクリーンかつ無効であることが要求される場合、キャッシュのクリーニング、またはクリーニングと無効化の命令のシーケンスが、割り込みが不可能でないときには常に割り込みの発生に対処できることが必要です。これは、以前にクリーニングされたキャッシュブロックに、割り込みによって書き込みが行われる可能性があるためです。このため、キャッシュダーティステータス レジスタは、最後にキャッシュのクリーニングが正常に行われた後で、キャッシュに書き込みが行われたかどうかを示します。

キャッシュダーティステータス レジスタを調べて、キャッシュがクリーンかどうかを判定できます。また、割り込みが不可能な間にこの操作を実行すると、以後の操作でキャッシュがクリーンであることを前提とできます。次のシーケンスは、この手法を示したものです。

```
; The following code assumes interrupts are enabled at this point.
Loop1
        MOV     R1, #0
        MCR     p15, 0, R1, c7, c10, 0          ; Clean data cache. For Clean and Invalidate,
                                                ;  use MCR p15, 0, R1, c7, c14, 0 instead
        MRS     R2, CPSR                        ; Save PSR context
        CPSID   iaf                             ; Disable interrupts
        MRC     p15, 0, R1, c7, c10, 6          ; Read Cache Dirty Status Register
        TST     R1, #1                          ; Check if it is clean
        BEQ     UseClean
        MSR     CPSR_xc, R2                     ; Re-enable interrupts
        B       Loop1                           ; Clean the cache again
UseClean
        Do_Clean_Operations                     ; Perform whatever operation relies on
                                                ;  the cache being clean or clean & invalid.
                                                ;  To reduce impact on interrupt latency,
                                                ;  this sequence should be short.
        MCR     p15, 0, R1, c7, c6, 0           ; Optional. Can use this Invalidate all command
                                                ;  to invalidate a Clean loop.
        MSR     CPSR_xc, R2                     ; Re-enable interrupts
```

注

長いキャッシュのクリーニング操作は、このルーチン全体を通して割り込みが可能な状態で実行されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Block transfer operations

ARMv7 does not support CP15 register block transfer operations, and they are optional in ARMv6. Table G-8 summarizes block transfer operations. Permitted combinations of the block transfer operations are:

- all four operations
- clean, clean and invalidate, and invalidate operations
- none of the operations.

If an operation is not implemented, then it must cause an Undefined Instruction exception.

Table G-8 Block transfer operations

Operation	Blocking[a] or non-blocking	Instruction or data	User or privileged	Exception Behavior
Prefetch range	Non-blocking	Instruction or data	User or privileged	None
Clean range	Blocking	Data only	User or privileged	Data Abort
Clean and Invalidate range	Blocking	Data only	Privileged	Data Abort
Invalidate range	Blocking	Instruction or data	Privileged	Data Abort

a. See *Blocking and non-blocking behavior* on page AppxG-42

An MCRR instruction starts each of the range operations. The data of the two registers specifies the Block start address and the Block end address. All block operations are performed on the cachelines that include the range of addresses between the Block start address and Block end address inclusive. If the Block start address is greater than the Block end address the effect is UNPREDICTABLE.

ARMv6 supports only one block transfer at a time. Attempting to start a second block transfer while a block transfer is in progress causes the first block transfer to be abandoned and starts the second block transfer. The Block Transfer Status Register indicates whether a block transfer is in progress. The register can be polled before starting a block transfer, to ensure any previous block transfer operation has completed.

All block transfers are interruptible. When blocking transfers are interrupted, the LR value that is captured is (address of instruction that launched the block operation + 4). This enables the standard return mechanism for interrupts to restart the operation.

For performance reasons, ARM recommends that implementations permit the following instructions to be executed while a non-blocking prefetch range instruction is being executed. In such an implementation, the LR value captured on an interrupt is determined by the instruction set state presented to the interrupt in the following instruction stream. However, implementations that treat a prefetch range instruction as a blocking operation must capture the LR value as described in the previous paragraph.

If the FCSE PID is changed while a prefetch range operation is running, it is UNPREDICTABLE at which point this change is seen by the prefetch range. For information about changing the FCSE PID see *c13, FCSE Process ID Register (FCSEIDR)* on page B3-152.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ブロック転送操作

ARMv7 では CP15 レジスタのブロック転送操作はサポートされておらず、ARMv6 ではオプションです。ブロック転送操作の要約を、表 G-8 に示します。ブロック転送操作の許される組み合わせは、次のいずれかです。

- 4 つの操作すべて
- クリーニング、クリーニングと無効化、および無効化操作
- いずれの操作もなし

操作が実装されていない場合は、未定義命令例外が発生する必要があります。

表 G-8 ブロック転送操作

操作	ブロッキング[a] または非ブロッキング	命令またはデータ	ユーザまたは特権	例外動作
範囲のプリフェッチ	非ブロッキング	命令またはデータ	ユーザまたは特権	なし
範囲のクリーニング	ブロッキング	データのみ	ユーザまたは特権	データアボート
範囲のクリーニングと無効化	ブロッキング	データのみ	特権	データアボート
範囲の無効化	ブロッキング	命令またはデータ	特権	データアボート

a. *ブロッキング動作と非ブロッキング動作* P. 付録 G-42 を参照して下さい。

MCRR 命令は、それぞれの範囲操作を開始します。2 つのレジスタのデータにより、ブロック開始アドレスおよびブロック終了アドレスが指定されます。すべてのブロック操作は、ブロック開始アドレスからブロック終了アドレスまでのアドレス範囲を含むキャッシュラインに対して実行されます。このアドレス範囲には、開始アドレスおよび終了アドレス自体も含まれます。ブロック開始アドレスがブロック終了アドレスよりも大きい場合、動作は予測不能です。

ARMv6 では、同時に 1 つのブロック転送のみがサポートされます。1 つ目のブロック転送の進行中に 2 つ目のブロック転送を開始しようとすると、最初のブロック転送は破棄され、2 つ目のブロック転送が開始されます。ブロック転送ステータスレジスタは、ブロック転送が進行中かどうかを示します。以前のブロック転送操作が完了しているかどうかを確認するには、ブロック転送を開始する前にこのレジスタをポーリングします。

すべてのブロック転送は割り込み可能です。ブロック転送で割り込みが発生した場合、取り込まれる LR 値は（ブロック操作を開始した命令のアドレス + 4）です。これにより、割り込みの標準的な復帰機構で操作を再開できます。

パフォーマンス上の理由から、実装では、非ブロッキングの範囲プリフェッチ命令の実行中に、後続の命令を実行可能にすることをお勧めします。このような実装の場合、割り込みで取り込まれる LR 値は、以後の命令ストリームで割り込みに示される命令セット状態によって決定されます。ただし、範囲プリフェッチ命令をブロッキング操作として処理する実装では、前の段落で説明したように LR 値を取り込む必要があります。

範囲プリフェッチ操作の実行中に FCSE PID が変更された場合、この変更がどの時点で範囲プリフェッチから可視になるかは予測不能です。FCSE PID の変更の詳細については、*c13、FCSE プロセス ID レジスタ (FCSEIDR)* P. B3-152 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Blocking and non-blocking behavior

The cache block transfer operations for cleaning, invalidating, or clean and invalidating a range of addresses from the cache are blocking operations. Following instructions must not be executed until the block transfer operation has completed. The prefetch range operation is non-blocking and can permit following instructions to be executed before the operation is complete. If an exception occurs a non-blocking operation does not signal an exception to the processor. This enables implementations to retire following instructions while the non-blocking operation is executing, without the requirement to retain precise processor state.

The blocking operations generate a Data Abort exception on a Translation fault if a valid translation table entry cannot be fetched. The DFAR indicates the address that caused the fault, and the DFSR indicates the reason for the fault.

Any fault on a prefetch range operation results in the operation failing without signaling an error.

Register encodings

Table G-9 shows the block operations supported using CP15. The operations are performed using an `MCRR` instruction. See *MCRR, MCRR2* on page A8-188.

The instruction format for block operations is:

```
MCRR p15, <Opc>, <Rt>, <Rn>, <CRm>
```

Table G-9 Enhanced cache control operations using MCRR

CRm	Opc	Function	Rn Data, VA [a]	Rt Data, VA [a]
c5	0	Invalidate instruction cache range [b]	Start address	End address
c6	0	Invalidate data cache range [b]	Start address	End address
c12	0	Clean data cache range [c]	Start address	End address
c12	1	Prefetch instruction range [c]	Start address	End address
c12	2	Prefetch data range [c]	Start address	End address
c14	0	Clean and invalidate data cache range [b]	Start address	End address

a. The true virtual address, before any modification by the FCSE. See Appendix E *Fast Context Switch Extension (FCSE)*. This address is translated by the FCSE logic.
b. Accessible only in privileged modes. Results in an UNDEFINED instruction exception if the operation is attempted in user mode.
c. Accessible in both unprivileged and privileged modes.

Note

The ARMv6 `MCRR` encodings that support block operations are UNDEFINED in ARMv7.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ブロッキング動作と非ブロッキング動作

キャッシュのアドレス範囲のクリーニング、無効化、またはクリーニングと無効化のキャッシュブロック転送操作は、ブロッキング操作です。後続の命令は、ブロック転送操作が完了するまで実行できません。範囲プリフェッチ操作は非ブロッキング操作で、操作が完了する前に後続の命令を実行できます。例外が発生した場合、非ブロッキング操作では例外はプロセッサに通知されません。これにより、実装は正確なプロセッサ状態を保持する必要なしに、非ブロッキング操作の実行中に後続の命令を終了できます。

ブロッキング操作では、有効な変換テーブルエントリをフェッチできない場合、変換フォールトに関するデータアボート例外が生成されます。DFAR はフォールトを引き起こしたアドレスを示し、DFSR はフォールトの原因を示します。

範囲プリフェッチ操作でフォールトが発生した場合、操作はエラーを通知せずに失敗します。

レジスタエンコード

CP15 を使用してサポートされるブロック操作を、表 G-9 に示します。これらの操作は、`MCRR` 命令を使用して実行されます。*MCRR*、*MCRR2* P. A8-188 を参照して下さい。

ブロック操作の命令フォーマットは次のとおりです。

```
MCRR p15, <Opc>, <Rt>, <Rn>, <CRm>
```

表 G-9 MCRR を使用した拡張キャッシュ制御操作

CRm	Opc	機能	Rn データ、VA [a]	Rt データ、VA [a]
c5	0	命令キャッシュ範囲の無効化[b]	開始アドレス	終了アドレス
c6	0	データキャッシュ範囲の無効化 [b]	開始アドレス	終了アドレス
c12	0	データキャッシュ範囲のクリーニング[c]	開始アドレス	終了アドレス
c12	1	命令範囲のプリフェッチ [c]	開始アドレス	終了アドレス
c12	2	データ範囲のプリフェッチ [c]	開始アドレス	終了アドレス
c14	0	データキャッシュ範囲のクリーニングと無効化 [b]	開始アドレス	終了アドレス

a. 真の仮想アドレス（FCSE によって変更が行われる前）。付録 E *「高速コンテキストスイッチ拡張機能 (FCSE)」*を参照して下さい。このアドレスは FCSE ロジックにより変換されます。
b. 特権モードでのみアクセス可能。ユーザモードでこの操作が試みられた場合、未定義命令例外が発生します。
c. 非特権モードと特権モードの両方でアクセス可能。

注

ブロック操作をサポートする ARMv6 `MCRR` エンコードは、ARMv7 では未定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The range operations operate on cache lines. The first cache line operated on is the line that contains the start address. The operation is then applied to every cache line up to and including the line that contains the end address.

The format of the start address and end address data values passed by the `MCRR` instructions is:

31 … L	L-1 … 0
Virtual address	Ignored

Start address **Virtual Address bits [31:L]**

The first virtual address of the block transfer.

End address **Virtual Address bits [31:L]**

The virtual address at which the block transfer stops. This address is at the start of the line containing the last address to be handled by the block transfer.

L is Log_2(LINELEN), where LINELEN is the cache line length parameter. Because the least significant address bits are ignored, the transfer automatically adjusts to a line length multiple spanning the programmed addresses.

Note

The block operations use virtual addresses, not modified virtual addresses. All other address-based cache operations use MVAs.

CP15 c7 operations for block transfer management

Two CP15 c7 operations support block transfer management. These operations must be implemented when the block transfer operations are implemented:

StopPrefetchRange `MCR p15, 0, <Rt>, c7, c12, 5; Write-only, <Rt> Should-Be-Zero`

PrefetchStatus `MRC p15, 0, <Rt>, c7, c12, 4; Read Block Transfer Status Register`

Both operations are accessible in unprivileged and privileged modes. Because all block operations are mutually exclusive, that is, only one operation can be active at any time, the PrefetchStatus operation returns the status of the last issued Prefetch request, instruction, or data. This status is held in the Block Transfer Status Register.

c7, Block Transfer Status Register

The Block Transfer Status Register indicates whether a block transfer is in progress.

The format of the Block Transfer Status Register is:

31 … 1	0
UNK	R

Bits [31:1] Reserved.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

範囲操作は、キャッシュラインに対して動作します。操作対象の最初のキャッシュラインは、開始アドレスが含まれるラインです。終了アドレスが含まれるラインまで、すべてのキャッシュラインにこの操作が適用されます。

`MCRR` 命令によって渡される開始アドレスと終了アドレスのデータ値のフォーマットは、次のとおりです。

31 … L	L－1 … 0
仮想アドレス	無視

開始アドレス **仮想アドレスビット [31:L]**

ブロック転送の最初の仮想アドレス。

終了アドレス **仮想アドレスビット [31:L]**

ブロック転送が停止する仮想アドレス。このアドレスは、ブロック転送によって処理される最後のアドレスを含むラインの先頭です。

L は、Log_2 (LINELEN) です。ここで、LINELEN はキャッシュライン長のパラメータです。最下位アドレスビットは無視されるため、転送の範囲はプログラムされたアドレス全体を含む、ライン長の倍数に自動的に調整されます。

注

ブロック操作では、修飾仮想アドレスではなく仮想アドレスが使用されます。他のすべてのアドレスベースのキャッシュ操作では MVA が使用されます。

ブロック転送管理用の CP15 c7 操作

2 つの CP15 c7 操作では、ブロック転送管理がサポートされます。ブロック転送操作が実装されている場合、これらの操作を実装する必要があります。

StopPrefetchRange `MCR p15, 0, <Rt>, c7, c12, 5; Write-only, <Rt> Should-Be-Zero`

PrefetchStatus `MRC p15, 0, <Rt>, c7, c12, 4; Read Block Transfer Status Register`

どちらの操作にも、非特権モードおよび特権モードでアクセス可能です。すべてのブロック操作は相互に排他的で、同時にアクティブとなれる操作は 1 つだけなので、PrefetchStatus 操作では、最後に発行されたプリフェッチ要求、命令、またはデータのステータスが返されます。このステータスは、ブロック転送ステータスレジスタに保持されます。

c7、ブロック転送ステータスレジスタ

ブロック転送ステータスレジスタは、ブロック転送が進行中かどうかを示します。

ブロック転送ステータスレジスタのフォーマットは次のとおりです。

31 … 1	0
UNK	R

ビット [31:1] 予約

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

R, bit [0] Block Prefetch Running

0 No prefetch in operation

1 Prefetch in operation.

G.7.11 c7, Miscellaneous functions

The Wait For Interrupt operation is used in some implementations as part of a power management support scheme. The operation is deprecated in ARMv6 and not supported in ARMv7, where it behaves as a `NOP` instruction.

Barrier operations are used for system correctness to ensure visibility of memory accesses to other agents in a system. Barrier functionality was formally defined as part of the memory architecture enhancements introduced in ARMv6. The definitions are the same as for ARMv7. For details see *Memory barriers* on page A3-47.

Table G-10 summarizes the `MCR` instruction encoding details.

Table G-10 memory barrier register support

Operation	CRn	opc1	CRm	opc2
Wait For Interrupt (CP15WFI)	c7	0	c0	4
Instruction Synchronization Barrier (CP15ISB)[a]	c7	0	c5	4
Data Synchronization Barrier (CP15DSB)[b]	c7	0	c10	4
Data Memory Barrier (CP15DMB)	c7	0	c10	5

a. This operation was previously known as *Prefetch Flush* (PF or PFF).
b. This operation was previously known as *Data Write Barrier* or *Drain Write Buffer* (DWB).

G.7.12 c7, VMSA virtual to physical address translation support

If the Security Extensions are implemented in ARMv6K, virtual to physical address translation support is provided as described in *CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130.

G.7.13 c8, VMSA TLB support

CP15 TLB operation provision in ARMv6 is the same as for ARMv7-A. For details see *TLB maintenance* on page B3-56 and *CP15 c8, TLB maintenance operations* on page B3-138.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

R、ビット [0] ブロックプリフェッチ実行中

0 プリフェッチが実行されていません。

1 プリフェッチが実行中です。

G.7.11 c7、その他の機能

割り込み待ち操作は、一部の実装で電力管理サポート方式の一部として使用されます。この操作は ARMv6 では非推奨で、ARMv7 ではサポートされず、この操作は `NOP` 命令として動作します。

バリア操作は、システムの正確性の目的で、システム内の他のエージェントからメモリアクセスが可視であることを保証するために使用されます。バリア機能は、ARMv6 で導入されたメモリアーキテクチャ拡張機能の一部として、正式に定義されています。この定義は、ARMv7 のものと同じです。詳細については、*メモリバリア* P. A3-47 を参照して下さい。

`MCR` 命令のエンコードの詳細を、表 G-10 に示します。

表 G-10 メモリバリア レジスタのサポート

動作	CRn	opc1	CRm	opc2
割り込み待ち (CP15WFI)	c7	0	c0	4
命令同期バリア (CP15ISB)[a]	c7	0	c5	4
データ同期バリア (CP15DSB)[b]	c7	0	c10	4
データメモリ バリア (CP15DMB)	c7	0	c10	5

a. この操作は、以前は*プリフェッチフラッシュ*（PF または PFF）と呼ばれていました。
b. この操作は、以前は*データ書き込みバリア*または*ライトバッファ ドレイン* (DWB) と呼ばれていました。

G.7.12 c7、VMSA の仮想アドレスから物理アドレスへの変換のサポート

ARMv6K にセキュリティ拡張機能が実装されている場合、*CP15 c7、仮想アドレスから物理アドレスへの変換操作* P. B3-130 で説明されているように、仮想アドレスから物理アドレスへの変換がサポートされます。

G.7.13 c8、VMSA TLB のサポート

ARMv6 での CP15 TLB 操作の機構は、ARMv7-A と同じです。詳細については、*TLB 保守* P. B3-56 および *CP15 c8、TLB 保守操作* P. B3-138 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.7.14 c9, Cache lockdown support

One problem with caches is that although they normally improve average access time to data and instructions, they usually increase the worst-case access time. This occurs for a number of reasons, including:

- There is a delay before the system determines that a cache miss has occurred and starts the main memory access.
- If a Write-Back cache is being used, there might be an extra delay because of the requirement to store the contents of the cache line that is being reallocated.
- A whole cache line is loaded from main memory, not only the data requested by the ARM processor.

In real-time applications, this increase in the worst-case access time can be significant.

Cache lockdown is an optional feature designed to alleviate this. It enables critical code and data, for example high priority interrupt routines and the data they access, to be loaded into the cache in such a way that the cache lines containing them are not subsequently reallocated. This ensures that all subsequent accesses to the code and data concerned are cache hits and therefore complete as quickly as possible.

From ARMv7, cache lockdown is IMPLEMENTATION DEFINED with no recommended formats or mechanisms on how it is achieved other than reserved CP15 register space. See *Cache lockdown* on page B2-8 and *CP15 c9, Cache and TCM lockdown registers and performance monitors* on page B3-141.

ARMv4 and ARMv5 specify four formats for the cache lockdown mechanism, known as Format A, Format B, Format C, and Format D. The Cache Type Register contains information on the lockdown mechanism adopted. See *c0, Cache Type Register (CTR)* on page AppxH-35. Formats A, B, and C all operate on cache ways. Format D is a cache entry locking mechanism.

ARMv6 cache lockdown support must comply with Format C or Format D. For more information, see *c9, cache lockdown support* on page AppxH-52.

Note

A Format D implementation must use the CP15 lockdown operations with the CRm == {c5,c6} encodings, and not the alternative encodings with CRm == {c1,c2}.

Interaction with CP15 c7 operations

Cache lockdown only prevents the normal replacement strategy used on cache misses from choosing to reallocate cache lines in the locked-down region. CP15 c7 operations that invalidate, clean, or clean and invalidate cache contents affect locked-down cache lines as normal. If invalidate operations are used, you must ensure that they do not use virtual addresses or cache set/way combinations that affect the locked-down cache lines. Otherwise, if it is difficult to avoid affecting the locked-down cache lines, repeat the cache lockdown procedure afterwards.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.7.14 c9、キャッシュロックダウンのサポート

キャッシュに関する問題の 1 つは、キャッシュにより一般にデータおよび命令への平均アクセス時間は短縮するが、ワーストケースのアクセス時間は増大するということです。これは、次のようないくつかの理由で発生します。

- キャッシュミスが発生したことをシステムが確認し、メインメモリへのアクセスを開始するまでに遅延が発生する。
- ライトバックキャッシュが使用されている場合、再割り当てされるキャッシュラインの内容を保管する必要があり、それによる余分な遅延が発生する可能性がある。
- ARM プロセッサによって要求されたデータだけではなく、1 つのキャッシュラインの全体がメインメモリからロードされる。

リアルタイムアプリケーションでは、ワーストケースでのアクセス時間の増加が問題になる場合があります。

キャッシュロックダウンは、これを軽減するために設計されたオプションの機能です。この機能を使用すると、例えば優先度の高い割り込みルーチンや、そのルーチンがアクセスするデータなどの重要なコードやデータを、それらが含まれるキャッシュラインが以後に再割り当てされないようにキャッシュにロードできます。これにより、対象のコードやデータに対する以降のアクセスが必ずキャッシュヒットになるため、可能な限り高速に実行されることが保証されます。

ARMv7 およびそれ以降では、キャッシュロックダウンは実装定義で、予約の CP15 レジスタ空間以外で実行される場合の推奨されるフォーマットおよび機構は存在しません。*キャッシュロックダウン* P. B2-8 および *CP15 c9、キャッシュおよび TCM ロックダウンレジスタとパフォーマンスモニタ* P. B3-141 を参照して下さい。

ARMv4 および ARMv5 では、キャッシュロックダウン機構に対して 4 つのフォーマットが指定され、これらはフォーマット A、フォーマット B、フォーマット C、フォーマット D と呼ばれます。キャッシュタイプ レジスタには、採用されているロックダウン機構に関する情報が含まれています。*c0、キャッシュタイプ レジスタ (CTR)* P. 付録 H-35 を参照して下さい。フォーマット A、B、C は、すべてキャッシュウェイで動作します。フォーマット D は、キャッシュエントリのロック機構です。

ARMv6 のキャッシュロックダウンのサポートは、フォーマット C またはフォーマット D に準拠する必要があります。詳細については、*c9、キャッシュロックダウンのサポート* P. 付録 H-52 を参照して下さい。

注

フォーマット D の実装では、CRm == {c5,c6} エンコードの CP15 ロックダウン操作を使用する必要があります。CRm == {c1,c2} の代替エンコードは使用できません。

CP15 c7 操作との相互動作

キャッシュロックダウンで回避されるのは、キャッシュミスで使用される通常の置き換え方式で、ロックダウンされたキャッシュラインを再割り当て用に選択することのみです。キャッシュ内容の無効化、クリーニング、またはクリーニングと無効化を行う CP15 c7 操作は、ロックダウンされたキャッシュラインにも、通常のラインと同様に影響を与えます。無効化操作を使用する場合は、その操作において、ロックダウンされたキャッシュラインに影響を与える仮想アドレスおよびキャッシュセット / ウェイの組み合わせが使用されていないことを確認する必要があります。ロックダウンされたキャッシュラインに影響を与えることを回避するのが困難である場合は、後でキャッシュロックダウンの手順を繰り返して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

G.7.15 c9, TCM support

In ARMv7, CP15 c9 encodings with CRm == {c0-c2,c5-c8} are reserved for IMPLEMENTATION DEFINED branch predictor, cache, and TCM operations. In ARMv6, the TCM Type Register can determine the TCM support the processor provides. See *c0, TCM Type Register (TCMTR)* on page AppxG-33. Table G-11 summarizes the additional register support for TCMs in ARMv6.

Table G-11 TCM register support

Instruction	TCM Register
`MRC|MCR p15, 0, <Rt>, c9, c1, 0`	Data TCM Region Register, DTCMRR
`MRC|MCR p15, 0, <Rt>, c9, c1, 1`	Instruction or unified TCM Region Register, ITCMRR
`MRC|MCR p15, 0, <Rt>, c9, c2, 0`	TCM Selection Register, TCMSR

Each implemented TCM has its own Region register that is banked onto either the Data TCM Region Register or the Instruction or unified TCM Region Register. The TCM Selection Register supplies the index for region register access.

Changing the TCM Region Register while a prefetch range or DMA operation is running has UNPREDICTABLE effects.

c9, TCM Selection Register (TCMSR)

The TCM Selection Register selects the current TCM Region Registers. Where separate data and instruction TCMs are implemented, the value in the TCM Selection Register defined the current region for accesses to both the Data TCM Region Register and the Instruction TCM Region Register, see Table G-11.

The TCM Selection Register is:

- a 32-bit read/write register
- accessible only in privileged modes
- when the Security Extensions are implemented, a Banked register.

The format of the TCM Selection Register is:

31 … 2	1	0
Reserved	TCM	

Bits [31:2] Reserved, UNK/SBZP.

TCM, bits [1:0]

TCM number, the index used to access a region register. TCM region registers can be accessed to read or change the details of the selected TCM.

This value resets to 0.

If this field is written with a value greater than or equal to the maximum number of implemented TCMs then the write is ignored.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

G.7.15 c9、TCM のサポート

ARMv7 では、CRm == {c0-c2,c5-c8} である CP15 c9 エンコードは、実装定義の分岐予測器、キャッシュ、TCM の操作用に予約されています。ARMv6 では、TCM タイプレジスタにより、プロセッサが提供する TCM サポートを決定できます。*c0、TCM タイプレジスタ (TCMTR)* P. 付録 G-33 を参照して下さい。ARMv6 における TCM 用の追加レジスタサポートの要約を、表 G-11 に示します。

表 G-11 TCM レジスタのサポート

命令	TCM レジスタ
`MRC|MCR p15, 0, <Rt>, c9, c1, 0`	データ TCM 領域レジスタ、DTCMRR
`MRC|MCR p15, 0, <Rt>, c9, c1, 1`	命令または統一 TCM 領域レジスタ、ITCMRR
`MRC|MCR p15, 0, <Rt>, c9, c2, 0`	TCM 選択レジスタ、TCMSR

実装されているそれぞれの TCM は、データ TCM 領域レジスタ、あるいは命令または統一 TCM 領域レジスタのいずれかにバンクされた独自の領域レジスタを持ちます。TCM 選択レジスタは、領域レジスタアクセス用のインデクスを提供します。

範囲のプリフェッチまたは DMA 操作の実行中に TCM 領域レジスタが変更された場合、影響は予測不能です。

c9、TCM 選択レジスタ (TCMSR)

TCM 選択レジスタは、現在の TCM 領域レジスタを選択します。データ TCM と命令 TCM が別々に実装されている場合は、TCM 選択レジスタの値により、データ TCM 領域レジスタおよび命令 TCM 領域レジスタの両方へのアクセスに使用される現在の領域が定義されます。表 G-11 を参照して下さい。

TCM 選択レジスタには、次の特徴があります。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、バンクレジスタです。

TCM 選択レジスタのフォーマットは次のとおりです。

31 … 2	1	0
予約	TCM	

ビット [31:2] 予約、UNK/SBZP

TCM、ビット [1:0]

TCM の番号。領域レジスタへのアクセスに使用されるインデクスです。選択した TCM の内容を読み出しまたは変更するため、TCM 領域レジスタにアクセスできます。

リセット時の値は 0 です。

実装されている TCM の最大数以上の値がこのフィールドに書き込まれた場合、その書き込みは無視されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c9, TCM Region Registers (DTCMRR and ITCMRR)

The TCM Region Registers provide control and configuration information for each TCM region.

Each TCM Region Register is:

- A 32-bit read/write register with some bits that are read-only.
- Accessible only in privileged modes.
- When the Security Extensions are implemented, a Configurable access register with Non-secure access controlled by the DTCM-NSACR. See *c9, TCM Non-Secure Access Control Registers, DTCM-NSACR and ITCM-NSACR* on page AppxG-51.
- Accessed by reading or writing the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c9, `<CRm>` set to c1, and `<opc2>` set to:
  - — 0 for the current Data TCM Region Register
  - — 1 for the current Instruction or unified Region Register.

  For example:

```
MRC p15,0,<Rt>,c9,c1,0    ; Read  current Data TCM Region Register
MCR p15,0,<Rt>,c9,c1,0    ; Write current Data TCM Region Register
MRC p15,0,<Rt>,c9,c1,1    ; Read  current Instruction or unified TCM Region Register
MCR p15,0,<Rt>,c9,c1,1    ; Write current Instruction or unified TCM Region Register
```

The format of the TCM region registers is:

31 … 12	11 … 7	6 … 2	1	0
BaseAddress	Reserved	Size	(0)	En

BaseAddress, bits [31:12]

The base address of the TCM, given as the physical address of the TCM in the memory map. BaseAddress is assumed to be aligned to the size of the TCM. Any address bits in the range [($\log_2$(RAMSize)-1):12] are ignored.

BaseAddress is 0 at reset.

Bits [11:7] Reserved

Size, bits [6:2]

Indicates the size of the TCM. See Table G-12 on page AppxG-48 for encoding of this field.

This field is read-only and ignores writes.

En, bit [0] TCM enable bit:

En == 0 Disabled. This is the reset value.

En == 1 Enabled.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c9、TCM 領域レジスタ (DTCMRR および ITCMRR)

TCM 領域レジスタは、各 TCM 領域の制御情報および構成情報を提供します。

各 TCM 領域レジスタには、次の特徴があります。

- 32 ビットの読み出し / 書き込みレジスタで、一部のビットは読み出し専用です。
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、非セキュアなアクセスを持つ構成可能なアクセスレジスタで、DTCM-NSACR によって制御されます。*c9、TCM 非セキュアアクセス制御レジスタ、DTCM-NSACR および ITCM-NSACR* P. 付録 G-51 を参照して下さい。
- `<opc1>` を 0、`<CRn>` を c9、`<CRm>` を c1、`<opc2>` を次の値に設定し、CP15 レジスタの読み出し / 書き込みによってアクセスします。
  - – 現在のデータ TCM 領域レジスタにアクセスするには、0
  - – 現在の命令または統一領域レジスタにアクセスするには、1

  次に例を示します。

```
MRC p15,0,<Rt>,c9,c1,0    ; Read  current Data TCM Region Register
MCR p15,0,<Rt>,c9,c1,0    ; Write current Data TCM Region Register
MRC p15,0,<Rt>,c9,c1,1    ; Read  current Instruction or unified TCM Region Register
MCR p15,0,<Rt>,c9,c1,1    ; Write current Instruction or unified TCM Region Register
```

TCM 領域レジスタのフォーマットは次のとおりです。

31 … 12	11 … 7	6 … 2	1	0
BaseAddress	予約	サイズ	(0)	En

BaseAddress、ビット [31:12]

TCM のベースアドレス。メモリマップの TCM の物理アドレスを指定します。BaseAddress は、TCM のサイズにアラインされていると見なされます。[($\log_2$(RAMSize) – 1):12] の範囲のアドレスビットはすべて無視されます。

リセット時の BaseAddress の値は 0 です。

ビット [11:7] 予約

サイズ、ビット [6:2]

TCM のサイズを示します。このフィールドのエンコードについては、P. 付録 G-48 の表 G-12 を参照して下さい。

このフィールドは読み出し専用で、書き込みは無視されます。

En、ビット [0] TCM イネーブルビット

En = 0 TCM は非稼動です。これはリセット時の値です。

En = 1 TCM は稼動しています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Note

Bit [1] was defined as a *SmartCache* enable bit in the previous version of the ARM architecture. SmartCache is now considered to be IMPLEMENTATION DEFINED and not documented in this manual.

Table G-12 shows the encoding of the Size field in the TCM Region Registers:

Table G-12 TCM size field encoding

Size field	Memory size
0b00000	0KByte
0b00001, 0b00010	Reserved
0b00011	4KByte
0b00100	8KByte
0b00101	16KByte
0b00110	32KByte
0b00111	64KByte
0b01000	128KByte
0b01001	256KByte
0b01010	512KByte
0b01011	1MByte
0b01100	2MByte
0b01101	4MByte
0b01110	8MByte
0b01111	16MByte
0b10000	32MByte
0b10001	64MByte
0b10010	128MByte
0b10011	256MByte
0b10100	512MByte
0b10101	1GByte

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

注

ARM アーキテクチャの以前のバージョンでは、ビット [1] は *SmartCache* イネーブルビットとして定義されています。現在では、SmartCache は実装定義と見なされ、本書では説明されていません。

TCM 領域レジスタのサイズフィールドのエンコードを、表 G-12 に示します。

表 G-12 TCM のサイズフィールドのエンコード

サイズフィールド	メモリサイズ
0b00000	0KB
0b00001、0b00010	予約
0b00011	4KB
0b00100	8KB
0b00101	16KB
0b00110	32KB
0b00111	64KB
0b01000	128KB
0b01001	256KB
0b01010	512KB
0b01011	1MB
0b01100	2MB
0b01101	4MB
0b01110	8MB
0b01111	16MB
0b10000	32MB
0b10001	64MB
0b10010	128MB
0b10011	256MB
0b10100	512MB
0b10101	1GB
0b10110	2GB
0b10111	4GB
0b11xxx	予約

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table G-12 TCM size field encoding (continued)

Size field	Memory size
0b10110	2GByte
0b10111	4GByte
0b11xxx	Reserved

An attempt to access a TCM region that is not implemented is UNPREDICTABLE. This can occur if the number of data and instruction TCMs supported is not the same.

The base address of each TCM must be different, and chosen so that no location in memory is contained in more than one TCM. If a location in memory is contained in more than one TCM, it is UNPREDICTABLE which memory location the instruction or data is returned from. Implementations must ensure that this situation cannot result in physical damage to the TCM.

G.7.16 c9, VMSA support for the Security Extensions

ARMv6K with VMSA support and the Security Extensions provides the following CP15 c9 support in addition to that defined for the Security Extensions in ARMv7-A:

- a Cache Behavior Override Register, CBOR
- where instruction TCM support is implemented, an ITCM Non-secure Access Control Register, ITCM_NSAC
- where data TCM support is implemented, a DTCM Non-secure Access Control Register, DTCM_NSAC.

c9, Cache Behavior Override Register (CBOR)

The Cache Behavior Override Register, CBOR, overrides some aspects of the normal cache behavior. Typically, these overrides are used for system debugging.

Note

Architecturally, the CBOR is defined only as part of the Security Extensions in ARMv6. It is IMPLEMENTATION DEFINED whether an ARMv7-A implementation includes the CBOR. An implementation that does not include the Security Extensions might implement the CBOR, but can implement only bits [2:0] of the register.

The CBOR is:

- a 32-bit read/write register
- accessible only in privileged modes

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

実装されていない TCM 領域へのアクセスの試みは予測不能です。これは、サポートされているデータ TCM と命令 TCM の数が等しくない場合に発生します。

各 TCM のベースアドレスは異なっている必要があり、同じメモリ位置が複数の TCM に含まれないように選択する必要があります。あるメモリ位置が複数の TCM に含まれている場合、命令またはデータがどのメモリ位置から返されるかは予測不能です。実装では、この状況により TCM が物理的な損傷を受けないことを保証する必要があります。

G.7.16 c9、セキュリティ拡張機能の VMSA サポート

VMSA サポートおよびセキュリティ拡張機能を持つ ARMv6K では、ARMv7-A のセキュリティ拡張機能用に定義されているサポートに加えて、次の CP15 c9 サポートが提供されます。

- キャッシュ動作オーバライドレジスタ、CBOR
- ITCM 非セキュアアクセス制御レジスタ、ITCM_NSAC（命令 TCM サポートが実装されている場合）
- DTCM 非セキュアアクセス制御レジスタ、DTCM_NSAC（データ TCM サポートが実装されている場合）

c9、キャッシュ動作オーバライドレジスタ (CBOR)

キャッシュ動作オーバライドレジスタ (CBOR) は、通常のキャッシュ動作に含まれる一部の要素をオーバライドします。通常、これらのオーバライドはシステムデバッグで使用されます。

注

CBOR はアーキテクチャ的に、ARMv6 ではセキュリティ拡張機能の一部としてのみ定義されます。ARMv7-A 実装に CBOR が含まれるかどうかは実装定義です。セキュリティ拡張機能が含まれていない実装では CBOR を実装できますが、レジスタのビット [2:0] のみを実装できます。

CBOR には、次の特徴があります。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能が実装されている場合、共通レジスタで、一部のビットにはセキュア状態でのみアクセス可能です。

CBOR のフォーマットは次のとおりです。

CBOR のリセット時の値は 0x00000000 です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- when the Security Extensions are implemented, a Common register, with some bits that can be accessed only in Secure state.

The format of the CBOR is:

The CBOR resets to `0x00000000`.

Register bits [5:3] are accessible only in Secure state. In Non-secure state they are RAZ/WI.

Bits [31:6] Reserved. UNK/SBZP.

S_WT, bit [5] Secure Write-Through. Controls whether Write-Through is forced for regions marked as Secure and Write-Back. The possible values of this bit are:

0 Do not force Write-Through. This corresponds to normal cache operation.

1 Force Write-Through for regions marked as Secure and Write-Back.

S_IL, bit [4] Secure instruction cache linefill. Can be used to disable instruction cache linefill for Secure regions. The possible values of this bit are:

0 Instruction cache linefill enabled. This corresponds to normal cache operation.

1 Instruction cache linefill disabled for regions marked as Secure.

S_DL, bit [3] Secure data cache linefill. Can be used to disable data cache linefill for Secure regions. The possible values of this bit are:

0 Data cache linefill enabled. This corresponds to normal cache operation.

1 Data cache linefill disabled for regions marked as Secure.

NS_WT, bit [2]

Non-secure Write-Through. Controls whether Write-Through is forced for regions marked as Non-secure and Write-Back. The possible values of this bit are:

0 Do not force Write-Through. This corresponds to normal cache operation.

1 Force Write-Through for regions marked as Non-secure and Write-Back.

NS_IL, bit [1] Non-secure instruction cache linefill. Can be used to disable instruction cache linefill for Non-secure regions. The possible values of this bit are:

0 Instruction cache linefill enabled. This corresponds to normal cache operation.

1 Instruction cache linefill disabled for regions marked as Non-secure.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

レジスタのビット [5:3] は、セキュア状態でのみアクセス可能です。非セキュア状態では、これらのビットは RAZ/WI です。

ビット [31:6] 予約。UNK/SBZP。

S_WT、ビット [5] セキュア ライトスルー。セキュアかつライトバックとしてマークされている領域に対して、ライトスルーを強制的に行うかどうかを制御します。このビットの可能な値は次のとおりです。

0 ライトスルーを強制的に行いません。これは通常のキャッシュ操作に相当します。

1 セキュアかつライトバックとしてマークされている領域に対して、ライトスルーを強制的に行います。

S_IL、ビット [4] セキュアな命令キャッシュラインフィル。セキュアな領域に対する命令キャッシュラインフィルを不可能にするために使用できます。このビットの可能な値は次のとおりです。

0 命令キャッシュラインフィルが可能です。これは通常のキャッシュ操作に相当します。

1 セキュアとしてマークされている領域に対する命令キャッシュラインフィルは不可能です。

S_DL、ビット [3] セキュアなデータキャッシュ ラインフィル。セキュアな領域に対するデータキャッシュ ラインフィルを不可能にするために使用できます。このビットの可能な値は次のとおりです。

0 データキャッシュ ラインフィルが可能です。これは通常のキャッシュ操作に相当します。

1 セキュアとしてマークされている領域に対するデータキャッシュ ラインフィルが不可能です。

NS_WT、ビット [2]

非セキュアなライトスルー。非セキュアかつライトバックとしてマークされている領域に対して、ライトスルーを強制的に行うかどうかを制御します。このビットの可能な値は次のとおりです。

0 ライトスルーを強制的に行いません。これは通常のキャッシュ操作に相当します。

1 非セキュアかつライトバックとしてマークされている領域に対して、ライトスルーを強制的に行います。

NS_IL、ビット [1] 非セキュアな命令キャッシュラインフィル。非セキュアな領域に対する命令キャッシュラインフィルを不可能にするために使用できます。このビットの可能な値は次のとおりです。

0 命令キャッシュラインフィルが可能です。これは通常のキャッシュ操作に相当します。

1 非セキュアとしてマークされている領域に対する命令キャッシュラインフィルが不可能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

NS_DL, bit [0]

Non-secure data cache linefill. Can be used to disable data cache linefill for Non-secure regions. The possible values of this bit are:

0 Data cache linefill enabled. This corresponds to normal cache operation.

1 Data cache linefill disabled for regions marked as Non-secure.

It might be necessary to ensure that cache contents are not changed, for example when debugging or when processing an interruptible cache operation. The CBOR provides this option.

For example, Clean All, and Clean and Invalidate All operations in Non-secure state might not prevent fast interrupts to the Secure side if the FW bit in the SCR is set to 0. In this case, operations in the Secure state can read or write Non-secure locations in the cache. Such operations might cause the cache to contain valid or dirty Non-secure entries after the Non-secure Clean All and Clean and Invalidate All operation has completed. To prevent this problem, the Secure state must be:

- prevented from allocating Non-secure entries into the cache by disabling Non-secure linefill
- made to treat all writes to Non-secure regions that hit in the cache as being write-though by forcing Non-secure Write-Through.

The CBOR provides separate controls for Secure and Non-secure memory regions, and can be used to prevent cache linefill, or to force Write-Through operation, while leaving the caches enabled. The controls for Secure memory regions can be accessed only when the processor is in the Secure state.

Accessing the CBOR

To access the CBOR you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c9, `<CRm>` set to c8, and `<opc2>` set to 0. For example:

```
MRC p15, 0, <Rt>, c9, c8, 0    ; Read  CP15 Cache Behavior Override Register
MCR p15, 0, <Rt>, c9, c8, 0    ; Write CP15 Cache Behavior Override Register
```

c9, TCM Non-Secure Access Control Registers, DTCM-NSACR and ITCM-NSACR

The *Data TCM Non-Secure Access Control Register* (DTCM-NSACR) defines the accessibility of the Data TCM Region Register when the processor is in Non-secure state.

The *Instruction TCM Non-Secure Access Control Register* (ITCM-NSACR) defines the accessibility of the current Instruction or Unified TCM Region Register when the processor is in Non-secure state.

For information on TCM support, see *Tightly Coupled Memory (TCM) support* on page AppxG-23.

The TCM-NSACR registers are:

- 32-bit read/write registers
- accessible only in privileged modes
- implemented only in a VMSAv6 implementation that includes the Security Extensions
- Secure registers, see *Restricted access CP15 registers* on page B3-73.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

NS_DL、ビット [0]

非セキュアなデータキャッシュ ラインフィル。非セキュアな領域に対するデータキャッシュ ラインフィルを不可能にするために使用できます。このビットの可能な値は次のとおりです。

0 データキャッシュ ラインフィルが可能です。これは通常のキャッシュ操作に相当します。

1 非セキュアとしてマークされている領域に対するデータキャッシュ ラインフィルが不可能です。

キャッシュの内容が変更されていないことの確認が必要となる場合があります。例えば、デバッグ時や割り込み可能なキャッシュ操作の処理中などです。CBOR ではこのオプションが提供されます。

例えば、非セキュア状態でキャッシュ全体のクリーニング、およびクリーニングと無効化の操作では、SCR の FW ビットが 0 にクリアされている場合、セキュア側での高速割り込みは回避されない可能性があります。この場合、セキュア状態での操作により、キャッシュの非セキュアな位置に読み出しまたは書き込みが行われることがあります。このような操作では、非セキュアな全体のクリーニング、およびクリーニングと無効化の操作が完了した後に、キャッシュに有効またはダーティな非セキュアエントリが含まれることになります。この問題を回避するには、セキュア状態で次の条件が満たされている必要があります。

- 非セキュアなラインフィルを不可能にして、非セキュアエントリのキャッシュへの割り当てが防止されている。
- 非セキュアなライトスルーを強制して、キャッシュにヒットする非セキュアな領域へのすべての書き込みを、ライトスルーとして処理する。

CBOR ではセキュアおよび非セキュアなメモリ領域に対する個別の制御が提供され、キャッシュを稼動状態に保ちながら、キャッシュラインフィルを回避するか、またはライトスルー操作を強制的するために使用できます。セキュアなメモリ領域の制御には、プロセッサがセキュア状態のときにのみアクセスできます。

CBOR へのアクセス

CBOR にアクセスするには、`<opc1>` を 0、`<CRn>` を c9、`<CRm>` を c8、`<opc2>` を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15, 0, <Rt>, c9, c8, 0    ; Read  CP15 Cache Behavior Override Register
MCR p15, 0, <Rt>, c9, c8, 0    ; Write CP15 Cache Behavior Override Register
```

c9、TCM 非セキュアアクセス制御レジスタ、DTCM-NSACR および ITCM-NSACR

データ TCM 非セキュアアクセス制御レジスタ(DTCM-NSACR) は、プロセッサが非セキュア状態の場合におけるデータ TCM 領域レジスタのアクセス可能性を定義します。

命令 TCM 非セキュアアクセス制御レジスタ(ITCM-NSACR) は、プロセッサが非セキュア状態の場合における現在の命令または統一 TCM 領域レジスタのアクセス可能性を定義します。

TCM サポートの詳細については、*密結合メモリ (TCM) のサポート* P. 付録 G-23 を参照して下さい。

TCM-NSACR レジスタには、次の特徴があります。

- 32 ビットの読み出し / 書き込みレジスタ
- 特権モードでのみアクセス可能
- セキュリティ拡張機能を含む VMSAv6 実装にのみ実装されている。
- セキュアレジスタです（*限定アクセス CP15 レジスタ* P. B3-73 を参照）。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The format of a TCM-NSACR register is:

Bits [31:1] Reserved, UNK/SBZP.

NS_access, bit [0]

Non-secure access. Defines the accessibility of the corresponding current TCM Region Register from the Non-secure state. The possible values of this bit are:

0 The corresponding TCM Region Register is accessible only in Secure privileged modes.

The information stored in the corresponding TCM is Secure, and the TCM is visible only if the processor is in the Secure state and the translation table is marked as Secure.

1 The corresponding TCM Region Register is accessible in privileged modes in both Secure and Non-secure state.

The information stored in the corresponding TCM is Non-secure. The TCM is visible in the Non-secure state. It is visible in the Secure state only if the translation table is marked correctly as Non-secure.

The value of the TCM-NSACR.NS_access bit and the processor security state determine whether the TCM is visible. The value of the NS bit for the translation table entry determines what data is visible in the TCM. Table G-13 shows when the TCM is visible, and what data is visible.

Table G-13 Visibility of TCM and TCM data

Processor security state	TCM-NSACR NS_access bit	Translation table NS value	TCM visibility	Data visible
Secure	0	0	Visible	Secure
Secure	0	1	Not visible	-
Secure	1	0	Not visible	-
Secure	1	1	Visible	Non-secure
Non-secure	0	x	Not visible	-
Non-secure	1	x	Visible	Non-secure

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

TCM-NSACR レジスタのフォーマットは次のとおりです。

ビット [31:1] 予約。UNK/SBZP。

NS_access、ビット [0]

非セキュアアクセス。対応する現在の TCM 領域レジスタに、非セキュア状態からアクセス可能かどうかを定義します。このビットの可能な値は次のとおりです。

0 対応する TCM 領域レジスタには、セキュア特権モードでのみアクセス可能です。

対応する TCM に格納されている情報はセキュアで、その TCM は、プロセッサがセキュア状態で、変換テーブルがセキュアとしてマークされている場合にのみ参照可能です。

1 対応する TCM 領域レジスタには、セキュアおよび非セキュア状態の両方の特権モードでアクセス可能です。

対応する TCM に格納されている情報は非セキュアです。TCM は、非セキュア状態で参照可能です。変換テーブルが非セキュアとして正しくマークされている場合にのみ、セキュア状態でも参照可能です。

TCM-NSACR.NS_access ビットの値およびプロセッサのセキュリティ状態により、TCM が参照可能であるかどうかが決定されます。変換テーブルエントリの NS ビットの値により、TCM の参照可能なデータが決定されます。TCM が参照可能な場合および参照可能なデータを、表 G-13 に示します。

表 G-13 TCM および TCM データの可視性

プロセッサのセキュリティ状態	TCM-NSACR の NS_access ビット	変換テーブルの NS の値	TCM の可視性	データの可視性
セキュア	0	0	可視	セキュア
セキュア	0	1	非可視	–
セキュア	1	0	非可視	–
セキュア	1	1	可視	非セキュア
非セキュア	0	x	非可視	–
非セキュア	1	x	可視	非セキュア

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table G-14 shows when the TCM Region Register can be accessed, permitting control of the TCM.

Table G-14 Accessibility of TCM Region Register

Processor security state	TCM-NSACR NS_access bit	TCM Region Register access
Secure	x	In privileged modes only
Non-secure	0	No access
Non-secure	1	In privileged modes only

Accessing the TCM-NSACR registers

To access the TCM-NSACR registers you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c9, `<CRm>` set to c1, and `<opc2>` set to:

- 2 to access the DTCM-NSACR register
- 3 to access the ITCM-NSACR register.

For example

```
MRC p15,0,<Rt>,c9,c1,2    ; Read  CP15 Data TCM Non-secure Access Control Register
MCR p15,0,<Rt>,c9,c1,2    ; Write CP15 Data TCM Non-secure Access Control Register
MRC p15,0,<Rt>,c9,c1,3    ; Read  CP15 Instruction TCM Non-secure Access Control Register
MCR p15,0,<Rt>,c9,c1,3    ; Write CP15 Instruction TCM Non-secure Access Control Register
```

G.7.17 c10, VMSA memory remapping support

ARMv7-A memory remapping is supported from ARMv6K with the addition of the SCTLR.TRE enable bit and two registers:

- the *c10, Primary Region Remap Register (PRRR)* on page B3-143
- the *c10, Normal Memory Remap Register (NMRR)* on page B3-146.

G.7.18 c10, VMSA TLB lockdown support

TLB lockdown is an optional feature that enables the results of specified translation table walks to be loaded into the TLB, in such a way that they are not overwritten by the results of subsequent translation table walks.

Translation table walks can take a long time, especially as they involve potentially slow main memory accesses. In real-time interrupt handlers, translation table walks caused by the TLB not containing translations for the handler or the data it accesses can increase interrupt latency significantly.

Two basic lockdown models are supported:

- a TLB lock by entry model
- a translate and lock model introduced as an alternative model in ARMv5TE.

From ARMv7-A, TLB lockdown is IMPLEMENTATION DEFINED with no recommended formats or mechanisms on how it is achieved other than reserved CP15 register space. See *TLB lockdown* on page B3-56 and *CP15 c10, Memory remapping and TLB control registers* on page B3-142.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

どのような場合に TCM 領域レジスタへのアクセスが可能で、TCM の制御が許可されるかを、表 G-14 に示します。

表 G-14 TCM 領域レジスタのアクセス可能性

プロセッサのセキュリティ状態	TCM-NSACR の NS_access ビット	TCM 領域レジスタへのアクセス
セキュア	x	特権モードのみ
非セキュア	0	アクセス不可
非セキュア	1	特権モードのみ

TCM-NSACR レジスタへのアクセス

TCM-NSACR レジスタにアクセスするには、`<opc1>` を 0、`<CRn>` を c9、`<CRm>` を c1、`<opc2>` を次の値に設定して、CP15 レジスタを読み出し / 書き込みします。

- DTCM-NSACR レジスタにアクセスする場合、2
- ITCM-NSACR レジスタにアクセスする場合、3

次に例を示します。

```
MRC p15,0,<Rt>,c9,c1,2    ; Read  CP15 Data TCM Non-secure Access Control Register
MCR p15,0,<Rt>,c9,c1,2    ; Write CP15 Data TCM Non-secure Access Control Register
MRC p15,0,<Rt>,c9,c1,3    ; Read  CP15 Instruction TCM Non-secure Access Control Register
MCR p15,0,<Rt>,c9,c1,3    ; Write CP15 Instruction TCM Non-secure Access Control Register
```

G.7.17 c10、VMSA メモリ再マッピングのサポート

ARMv7-A のメモリ再マッピングは ARMv6K およびそれ以降でサポートされており、SCTLR.TRE イネーブルビットおよび次の 2 つのレジスタが追加されています。

- *c10、1 次領域再マップレジスタ (PRRR)* P. B3-143
- *c10、ノーマルメモリ再マップレジスタ (NMRR)* P. B3-146

G.7.18 c10、VMSA TLB ロックダウンのサポート

TLB ロックダウンはオプション機能で、指定した変換テーブルウォークの結果を TLB にロードできます。この場合、以後の変換テーブルウォークの結果で上書きされることはありません。

変換テーブルウォークは、低速な可能性があるメインメモリ アクセスが含まれている場合は特に、長時間を要する場合があります。リアルタイム割り込みハンドラでは、アクセスするハンドラまたはデータの変換が含まれていない TLB によって引き起こされる変換テーブルウォークにより、割り込みレイテンシが大幅に増加する場合があります。

2 つの基本的なロックダウンモデルがサポートされます。

- エントリモデルによる TLB ロック
- ARMv5TE で代替モデルとして導入された変換およびロックモデル

ARMv7-A 以降では、TLB ロックダウンは実装定義であり、予約 CP15 レジスタ空間以外で実行される場合の推奨フォーマットおよび機構は存在しません。*TLB ロックダウン* P. B3-56 および *CP15 c10、メモリ再マッピングおよび TLB 制御レジスタ* P. B3-142 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

For ARMv6, TLB lockdown must comply with one of the lockdown models described in *c10, VMSA TLB lockdown support* on page AppxH-59.

G.7.19 c11, DMA support

The ARMv6 DMA support for TCMs described in *The ARM Architecture Reference Manual* (DDI 0100) is considered IMPLEMENTATION DEFINED and not included in this manual. ARMv6 is therefore the same as ARMv7. See *CP15 c11, Reserved for TCM DMA registers* on page B3-147.

G.7.20 c12, VMSA support for the Security Extensions

CP15 c12 support for the Security Extensions in ARMv6 is the same as in ARMv7:

- the Vector Base Address Register, VBAR
- the Monitor Base Address Register, MVBAR
- the Interrupt Status Register, ISR.

For details see *CP15 c12, Security Extensions registers* on page B3-148.

G.7.21 c13, Context ID support

Both PMSAv6 and VMSAv6 require the CONTEXTIDR described in:

- *c13, Context ID Register (CONTEXTIDR)* on page B3-153, for a VMSA implementation
- *c13, Context ID Register (CONTEXTIDR)* on page B4-76, for a PMSA implementation.

In addition:

- A VMSAv6 implementation requires the FCSEIDR, described in *c13, FCSE Process ID Register (FCSEIDR)* on page B3-152. In ARMv6 the FCSE must be implemented. For more information, see Appendix E *Fast Context Switch Extension (FCSE)*.
- An ARMv6K implementation requires the Software Thread ID registers described in *CP15 c13 Software Thread ID registers* on page B3-154.

Note

In ARMv6, after any change to the CONTEXTIDR or FCSEIDR, software must use the CP15 branch predictor maintenance operations to flush the virtual addresses affected by the change. If the branch predictor is not invalidated in this way, attempting to execute an old branch might cause UNPREDICTABLE behavior. ARMv7 does not require branch predictors to be invalidated after a change to the CONTEXTIDR or FCSEIDR.

G.7.22 c15, IMPLEMENTATION DEFINED

As in ARMv7, CP15 c15 is reserved for IMPLEMENTATION DEFINED use. Typically, it is used for processor-specific runtime and test features.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ARMv6 では、TLB ロックダウンは、*c10、VMSA TLB ロックダウンのサポート* P. 付録 H-59 で説明されているロックダウンモデルのいずれかに準拠している必要があります。

G.7.19 c11、DMA のサポート

『*ARM アーキテクチャ リファレンスマニュアル*』(DDI 0100) で説明されている、TCM に対する ARMv6 DMA のサポートは実装定義と見なされ、本書では説明されていません。このため、ARMv6 では ARMv7 の場合と同じです。*CP15 c11、TCM DMA 用に予約されているレジスタ* P. B3-147 を参照して下さい。

G.7.20 c12、セキュリティ拡張機能の VMSA サポート

ARMv6 でのセキュリティ拡張機能に対する CP15 c12 サポートは、ARMv7 と同じです。

- ベクタベースアドレス レジスタ、VBAR
- モニタベースアドレス レジスタ、MVBAR
- 割り込みステータスレジスタ、ISR

詳細については、*CP15 c12、セキュリティ拡張機能レジスタ* P. B3-148 を参照して下さい。

G.7.21 c13、コンテキスト ID のサポート

PMSAv6 および VMSAv6 では、次のセクションで説明されている CONTEXTIDR が必要です。

- *c13、コンテキスト ID レジスタ (CONTEXTIDR)* P. B3-153（VMSA 実装の場合）
- *c13、コンテキスト ID レジスタ (CONTEXTIDR)* P. B4-76（PMSA 実装の場合）

これらに加えて、次の規則が適用されます。

- VMSAv6 実装には、*c13、FCSE プロセス ID レジスタ (FCSEIDR)* P. B3-152 で説明されている FCSEIDR が必要です。ARMv6 では、FCSE を実装する必要があります。詳細については、付録 E *「高速コンテキストスイッチ拡張機能 (FCSE)」* を参照して下さい。
- ARMv6K 実装には、*CP15 c13、ソフトウェアスレッド ID レジスタ* P. B3-154 で説明されているソフトウェアスレッド ID レジスタが必要です。

注

ARMv6 では、CONTEXTIDR または FCSEIDR への変更後に、ソフトウェアで CP15 分岐予測器保守操作を使用して、変更によって影響を受ける仮想アドレスをフラッシュする必要があります。この方法で分岐予測器が無効化されていない場合、以前の分岐を実行しようとすると予測不能な動作を引き起こす場合があります。ARMv7 では、CONTEXTIDR または FCSEIDR への変更後に分岐予測器を無効化する必要はありません。

G.7.22 C15、実装定義

ARMv7 と同様に、CP15 c15 は実装定義の使用に予約されています。通常、これはプロセッサ固有の実行時およびテストの機能で使用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Appendix H
ARMv4 and ARMv5 Differences

This appendix describes how the ARMv4 and ARMv5 architectures differ from the ARMv6 and ARMv7 architectures. It contains the following sections:

- *Introduction to ARMv4 and ARMv5* on page AppxH-2
- *Application level register support* on page AppxH-4
- *Application level memory support* on page AppxH-6
- *Instruction set support* on page AppxH-11
- *System level register support* on page AppxH-18
- *System level memory model* on page AppxH-21
- *System Control coprocessor (CP15) support* on page AppxH-31.

Note

In this appendix, the description ARMvN refers to all architecture variants of ARM architecture vN. For example, ARMv4 refers to all architecture variants of ARMv4, including ARMv4 and ARMv4T.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

付録 H
ARMv4 および ARMv5 での相違点

この付録では、ARMv4 および ARMv5 アーキテクチャが ARMv6 および ARMv7 アーキテクチャと異なる点について説明します。本章は次のセクションから構成されています。

- *ARMv4 および ARMv5 の概要* P. 付録 H-2
- *アプリケーションレベルのレジスタサポート* P. 付録 H-4
- *アプリケーションレベルのメモリサポート* P. 付録 H-6
- *命令セットのサポート* P. 付録 H-11
- *システムレベル レジスタのサポート* P. 付録 H-18
- *システムレベルのメモリモデル* P. 付録 H-21
- *システム制御コプロセッサ (CP15) のサポート* P. 付録 H-31

注

この付録で、ARMvN という記述は、ARM アーキテクチャ vN のすべてのアーキテクチャバリアントを意味します。例えば、ARMv4 は、ARMv4 および ARMv4T を含む ARMv4 のすべてのアーキテクチャバリアントを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.1 Introduction to ARMv4 and ARMv5

ARMv4 and ARMv5 defined the instruction set support and the programmers' model that applies to the general-purpose registers and the associated exception model. These architecture versions are fully described in the *ARM Architecture Reference Manual (DDI 0100)*.

Note

This appendix is a summary of the ARMv4 and ARMv5 architecture variants. It is expected that the majority of requirements for architecture information on ARMv4 and ARMv5 are satisfied by this appendix and the rest of this manual. However the *ARM Architecture Reference Manual* (DDI 0100) might be required for more information specific to ARMv4 or ARMv5.

Memory support is IMPLEMENTATION DEFINED in ARMv4 and ARMv5. In practice, use of CP15 to support the *Virtual Memory System Architecture* (VMSA) or *Protected Memory System Architecture* (PMSA) is standard in ARMv4 and ARMv5 implementations, but this is not an architectural requirement. For this reason, the datasheet or Technical Reference Manual for a particular ARM processor is the definitive source for its memory and system control facilities. This appendix does not specify absolute requirements on the functionality of CP15 or other memory system components. Instead, it contains guidelines designed to maximize compatibility with current and future ARM software.

This appendix concentrates on the features supported in ARMv4 and ARMv5, highlighting:

- features common across all architecture variants
- features supported for legacy reasons in ARMv6, but not in ARMv7
- features unique to the ARMv4 and ARMv5 variants.

H.1.1 Debug

Debug is not architecturally-defined in ARMv4 or ARMv5. ARM implementations have traditionally supported halting debug through a JTAG port. While the support of debug features is similar across ARM implementations, the timing and control sequencing required for access varies. Debug support in ARMv4 and ARMv5 is microarchitecture dependent and so in architectural terms is IMPLEMENTATION DEFINED.

H.1.2 ARMv6 and ARMv7

The ARM architecture was extended considerably in ARMv6. This means that a large proportion of this manual does not apply to earlier architecture variants and can be ignored with respect to ARMv4 and ARMv5.

The key changes in ARMv6:

- add:
  - the ARM SIMD instructions to improve execution of multimedia and other DSP applications
  - instructions for improved context switching.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

H.1 ARMv4 および ARMv5 の概要

ARMv4 および ARMv5 では、命令セットサポート、汎用レジスタに適用されるプログラマモデル、および関連する例外モデルが定義されます。これらのアーキテクチャバージョンについては、『*ARM アーキテクチャ リファレンスマニュアル*』(DDI 0100) で詳細に説明されています。

注

この付録は、ARMv4 および ARMv5 アーキテクチャバリアントの要約です。ARMv4 および ARMv5 に関するアーキテクチャ情報の要件の大部分が、この付録および本書の他の部分にある説明で満たされることが想定されています。ただし、ARMv4 または ARMv5 に固有の詳細情報については、『*ARM アーキテクチャ リファレンスマニュアル*』(DDI 0100) が必要になる場合があります。

ARMv4 および ARMv5 では、メモリサポートは実装定義です。実際、ARMv4 および ARMv5 実装では、*仮想メモリシステム アーキテクチャ*(VMSA) または*保護メモリシステム アーキテクチャ*(PMSA) をサポートするために CP15 の使用が標準的ですが、これはアーキテクチャ上の要件ではありません。この理由から、特定の ARM プロセッサのデータシートまたはテクニカルリファレンスマニュアルは、メモリおよびシステム制御機能に関する最も確実なソースになります。この付録では、CP15 または他のメモリシステム コンポーネントの機能に関する絶対的な要件は指定されていません。代わりに、現在および将来の ARM ソフトウェアと最大の互換性を持たせるように設計されたガイドラインが含まれています。

この付録では、ARMv4 および ARMv5 でサポートされる機能に重点を置き、次の点を中心に説明します。

- すべてのアーキテクチャバリアント間で共通な機能
- ARMv6 で従来と同様にサポートされ、ARMv7 ではサポートされない機能
- ARMv4 および ARMv5 バリアントに固有の機能

H.1.1 デバッグ

ARMv4 または ARMv5 では、デバッグはアーキテクチャ上は定義されていません。ARM 実装では、JTAG ポートを介したホールトデバッグが従来からサポートされています。それぞれの ARM 実装間でデバッグ機能のサポートは類似していますが、アクセスに必要なタイミングおよび制御シーケンスは異なります。ARMv4 および ARMv5 におけるデバッグのサポートはマイクロアーキテクチャ依存であるため、アーキテクチャ用語では実装定義となります。

H.1.2 ARMv6 および ARMv7

ARM アーキテクチャは、ARMv6 で大幅に拡張されています。これは、本書のかなりの部分が以前のアーキテクチャバリアントには適用されず、ARMv4 および ARMv5 に関しては無視できることを意味します。

ARMv6 での主要な変更点は次のとおりです。

- 次の命令が追加されています。
  - マルチメディアおよび他の DSP アプリケーションの実行を改良するための ARM SIMD 命令
  - コンテキストスイッチを改良する命令

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- introduce:
  - — a formal memory model, including level 1 cache support and revisions to alignment and endian support
  - — a requirement to provide either a VMSA or a PMSA for memory management
  - — a formal debug model
  - — a requirement to support the CP15 System Control coprocessor
  - — enhanced kernel support, in ARMv6K
  - — the optional Security Extensions
  - — 32-bit Thumb instructions, in ARMv6T2.

For information about the changes between ARMv6 and ARMv7 see Appendix G *ARMv6 Differences*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- 次のものが導入されています。
  - – 正式なメモリモデル（レベル 1 キャッシュサポートおよびアライメントとエンディアンサポートのリビジョンを含む）
  - – メモリ管理用に VMSA または PMSA のいずれかを提供する要件
  - – 正式なデバッグモデル
  - – CP15 システム制御コプロセッサをサポートする要件
  - – 拡張カーネルのサポート（ARMv6K の場合）
  - – セキュリティ拡張機能（オプション）
  - – 32 ビット Thumb 命令（ARMv6T2 の場合）

ARMv6 と ARMv7 との間の変更点の詳細については、付録 G「*ARMv6 との相違点*」を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.2 Application level register support

The ARMv4 and ARMv5 core registers are the same as ARMv7. For more information, see *ARM core registers* on page A2-11. The following sections give more information about ARMv4 and ARMv5 application level register support:

- *APSR support*
- *Instruction set state*.

H.2.1 APSR support

Program status is reported in the 32-bit *Application Program Status Register* (APSR). The format of the APSR is:

31	30	29	28	27	26	24	23	0
N	Z	C	V	Q	RAZ/SBZP		Reserved	

For details of the bit definitions, see *The Application Program Status Register (APSR)* on page A2-14. In the APSR descriptions:

- the GE[3:0] field is only defined from ARMv6, and is reserved in ARMv4 and ARMv5
- the Q bit is only defined from ARMv5TE, and is RAZ/WI in ARMv4, ARMv4T and ARMv5T.

Earlier versions of this manual do not use the term APSR. They refer to the APSR as the CPSR with the restriction on reserved fields governed by whether the register access was privileged.

H.2.2 Instruction set state

The instruction set states available in ARMv4 and ARMv5 are a subset of the states supported in ARMv7. All implementations support the ARM instruction set that executes in ARM state. All ARM instructions are 32-bit instructions. T variants of the architecture also support a 16-bit instruction set that executes in Thumb state. The supported ARM and Thumb instructions are summarized in *Instruction set support* on page AppxH-11.

Instruction set state support in ARMv4 and ARMv5 differs from the support available in ARMv7 as follows:

- ThumbEE state is not supported
- Jazelle state is supported only in ARMv5TEJ
- In privileged modes, you must take care not to attempt to change the instruction set state by writing nonzero values to CPSR.J and CPSR.T with an `MSR` instruction. For more information, see *Format of the CPSR and SPSRs* on page AppxG-17.

All ARMv4 and ARMv5 implementations support the ARM instruction set. ARMv4T, ARMv5T, ARMv5TE, and ARMv5TEJ also support a subset of the Thumb instruction set that can be executed entirely as 16-bit instructions. The only 32-bit instructions in this subset are restricted-range versions of the `BL` and `BLX` (immediate) instructions. See *BL and BLX (immediate) instructions, before ARMv6T2* on page AppxG-4 for a description of how these instructions can be executed as 16-bit instructions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

H.2 アプリケーションレベルのレジスタサポート

ARMv4 および ARMv5 のコアレジスタは、ARMv7 と同じです。詳細については、*ARM コアレジスタ* P. A2-11 を参照して下さい。次に示すセクションでは、ARMv4 および ARMv5 におけるアプリケーションレベルのレジスタサポートの詳細について説明します。

- *APSR のサポート*
- *命令セット状態*

H.2.1 APSR のサポート

プログラムステータスは、32 ビットの *アプリケーションプログラム ステータスレジスタ* (APSR) で報告されます。APSR のフォーマットは次のとおりです。

31	30	29	28	27	26	24	23	0
N	Z	C	V	Q	RAZ/SBZP		予約	

ビット定義の詳細については、*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。APSR の記述の項目には、次の条件があります。

- GE[3:0] フィールドは ARMv6 およびそれ以降でのみ定義され、ARMv4 および ARMv5 では予約です。
- Q ビットは ARMv5TE およびそれ以降でのみ定義され、ARMv4、ARMv4T、ARMv5T では RAZ/WI です。

本書の初期のバージョンでは、APSR という用語は使用されていません。初期のバージョンでは APSR は CPSR と呼ばれ、レジスタへのアクセスが特権付きであるかどうかによって予約フィールドが管理されるという制限が付いています。

H.2.2 命令セット状態

ARMv4 および ARMv5 で使用可能な命令セット状態は、ARMv7 でサポートされる状態のサブセットです。ARM 状態で実行される ARM 命令セットは、すべての実装でサポートされます。ARM 命令はすべて 32 ビット命令です。アーキテクチャの T バリアントでは、Thumb 状態で実行される 16 ビット命令セットもサポートされます。サポートされる ARM 命令および Thumb 命令の要約を、*命令セットのサポート* P. 付録 H-11 に示します。

ARMv4 および ARMv5 における命令セット状態のサポートは、ARMv7 で使用可能なサポートとは、次のように異なります。

- ThumbEE 状態はサポートされません。
- Jazelle 状態は ARMv5TEJ でのみサポートされます。
- 特権モードでは、`MSR` 命令により CPSR.J および CPSR.T に 0 以外の値を書き込んで命令セット状態の変更を試みないように注意する必要があります。詳細については、*CPSR と SPSR のフォーマット* P. 付録 G-17 を参照して下さい。

すべての ARMv4 および ARMv5 実装は、ARM 命令セットをサポートします。ARMv4T、ARMv5T、ARMv5TE、ARMv5TEJ は、完全に 16 ビット命令として実行できる Thumb 命令セットのサブセットもサポートします。このサブセットに含まれる 32 ビットは、`BL` および `BLX`（イ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Instruction set support on page AppxH-11 summarizes the ARM and Thumb instructions supported in ARMv4 and ARMv5, and the instruction descriptions in Chapter A8 *Instruction Details* give details of the architecture variants that support each instruction encoding.

Interworking

In ARMv4T, the only instruction that supports interworking branches between ARM and Thumb states is `BX`.

In ARMv5T, the `BLX` instruction was added to provide interworking procedure calls. The `LDR`, `LDM` and `POP` instructions were modified to perform interworking branches if they load a value into the PC. This is described by the `LoadWritePC()` pseudocode function. See *Pseudocode details of operations on ARM core registers* on page A2-12.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ミディエート）命令の範囲制限のあるバージョンのみです。これらの命令を 16 ビット命令として実行する方法の説明については、*ARM v 6T2 以前の BL および BLX（イミディエート）命令* P. 付録 G-4 を参照して下さい。

ARMv4 および ARMv5 でサポートされる ARM 命令および Thumb 命令の要約を、*命令セットのサポート* P. 付録 H-11 に示します。また、第 A8 章 *「命令の詳細」*の命令の説明では、各命令エンコードをサポートするアーキテクチャバリアントの詳細が示されています。

インターワーキング

ARMv4T では、ARM 状態と Thumb 状態との間のインターワーキング分岐をサポートする命令は `BX` のみです。

ARMv5T では、インターワーキングプロシージャ呼び出しを提供するために `BLX` 命令が追加されています。`LDR`、`LDM`、および `POP` 命令は、値を PC にロードする場合にインターワーキング分岐を実行するように変更されています。この動作は、`LoadWritePC()` 擬似コード関数で記述されます。詳細については、*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.3 Application level memory support

Memory support covers address alignment, endian support, semaphore support, memory type, memory order model, caches, and write buffers.

H.3.1 Alignment

ARMv4 and ARMv5 behave differently from ARMv7 for unaligned memory accesses. The behavior is the same as ARMv6 legacy mode except for forcing alignment checks with SCTLR.A == 1.

For more information about the SCTLR see *c1, System Control Register (SCTLR)* on page AppxH-39.

For ARM instructions when SCTLR.A == 0:

- Non halfword-aligned LDRH, LDRSH, and STRH are UNPREDICTABLE.
- Non word-aligned LDR, LDRT, and the load access of a SWP rotate right the word-aligned data transferred by a non word-aligned address one, two, or three bytes depending on the value of address[1:0].
- Non word-aligned STR, STRT, and the store access of a SWP ignore address[1:0].
- From ARMv5TE, it is IMPLEMENTATION DEFINED whether LDRD and STRD must be doubleword-aligned or word-aligned. LDRD and STRD instructions that do not meet the alignment requirement are UNPREDICTABLE.
- Non word-aligned LDM, LDC, LDC2, and POP ignore address[1:0].
- Non word-aligned STM, STC, STC2, and PUSH ignore address[1:0].

For Thumb instructions when SCTLR.A == 0:

- Non halfword-aligned LDRH, LDRSH, and STRH are UNPREDICTABLE.
- Non word-aligned LDR, and STR are UNPREDICTABLE.
- Non word-aligned LDMIA, and POP ignore address[1:0].
- Non word-aligned STMIA, and PUSH ignore address[1:0].

For ARM and Thumb instructions, alignment checking is defined for implementations supporting CP15, specifically the SCTLR.A bit. When this bit is set, a Data Abort exception indicating an Alignment fault is generated for unaligned accesses. When SCTLR.A = 1, whether the alignment check for an LDRD or STRD instruction is for doubleword-alignment or word-alignment depends on the implementation choice of which alignments are supported for these instructions when SCTLR.A = 0.

Note

The option of word alignment for LDRD and STRD instructions is not permitted in the ARMv6 legacy configuration where SCTLR.U == 0 and SCTLR.A == 1. For more information, see legacy alignment support in *Alignment* on page AppxG-6.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

H.3 アプリケーションレベルのメモリサポート

メモリサポートでは、アドレスアライメント、エンディアンサポート、セマフォサポート、メモリタイプ、メモリオーダ モデル、キャッシュ、ライトバッファが扱われます。

H.3.1 アライメント

ARMv4 および ARMv5 のアンアラインド メモリアクセスは、ARMv7 の場合とは異なる動作をします。この動作は、SCTLR.A == 1 で強制的にアライメントチェックを行うことを除いて、ARMv6 の従来のモードと同じです。

SCTLR の詳細については、*c1、システム制御レジスタ (SCTLR)* P. 付録 H-39 を参照して下さい。

ARM 命令で SCTLR.A == 0 の場合は、次の規則が適用されます。

- 非ハーフワードアラインドの LDRH、LDRSH、STRH は予測不能です。
- 非ワードアラインドの LDR、LDRT、および SWP のロードアクセスでは、アドレス [1:0] の値に応じて、非ワードアラインドアドレスによって転送されたワードアラインドデータが、1、2、または 3 バイトだけ右ローテートされます。
- 非ワードアラインドの STR、STRT、および SWP のストアアクセスでは、アドレス [1:0] は無視されます。
- ARMv5TE およびそれ以降では、LDRD および STRD がダブルワードアラインドの必要があるか、ワードアラインドの必要があるかは実装定義です。アライメントの要件を満たさない LDRD および STRD 命令は予測不能です。
- 非ワードアラインドの LDM、LDC、LDC2、POP では、アドレス [1:0] は無視されます。
- 非ワードアラインドの STM、STC、STC2、PUSH では、アドレス [1:0] は無視されます。

Thumb 命令で SCTLR.A == 0 の場合は、次の規則が適用されます。

- 非ハーフワードアラインドの LDRH、LDRSH、STRH は予測不能です。
- 非ワードアラインドの LDR および STR は予測不能です。
- 非ワードアラインドの LDMIA および POP では、アドレス [1:0] は無視されます。
- 非ワードアラインドの STMIA および PUSH では、アドレス [1:0] は無視されます。

ARM 命令および Thumb 命令の場合、アライメントチェックは CP15、具体的には SCTLR.A ビットをサポートする実装について定義されます。このビットがセットされているときは、アンアラインドアクセスについてアライメントフォールトを示すデータアボート例外が生成されます。SCTLR.A = 1 の場合、LDRD または STRD 命令に対するアライメントチェックの対象がダブルワードアライメントであるかワードアライメントであるかは、SCTLR.A = 0 の際にこれらの命令について、実装でどのアライメントのサポートが選択されているかによって異なります。

注

LDRD および STRD 命令のワードアライメントのオプションは、SCTLR.U == 0 かつ SCTLR.A == 1 である ARMv6 の従来の構成では許可されません。詳細については、*アライメント* P. 付録 G-6 にある、従来のアライメントサポートを参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.3.2 Endian support

ARMv4 and ARMv5 support big and little endian operation. Little endian support is consistent with ARMv7. Big endian control, configuration, and the connectivity of data bytes between the ARM register file and memory is different. However, the difference is only visible when communicating between big endian and little endian agents using memory. The agents can be different processors or programs running with different endianness settings on the same processor.

For ARMv4 and ARMv5, the distinction between big endian memory and little endian memory is managed by changing the addresses of the bytes in a word. For ARMv7, the distinction between big endian memory and little endian memory is managed by keeping the byte addresses the same, and reordering the bytes in the halfword or word. The endian formats are:

LE Little endian format used by ARMv4, ARMv5, ARMv6, and ARMv7

BE Big endian format used by ARMv6 (endianness controlled by the `SETEND` instruction) and ARMv7

BE-32 Big endian format used by ARMv4, ARMv5, and ARMv6 (legacy format, endianness controlled by the SCTLR.B bit).

Table H-1 shows how the addresses of bytes are changed in the BE-32 endian format. In this table, A is a doubleword-aligned address and S, T, U, V, W, X, Y, Z are the bytes at addresses A to A+7 in the ARMv7 memory map.

Table H-1 Addresses of bytes in endian formats

Byte	Address in format BE or LE	Address in format BE-32
S	A	A+3
T	A+1	A+2
U	A+2	A+1
V	A+3	A
W	A+4	A+7
X	A+5	A+6
Y	A+6	A+5
Z	A+7	A+4

Aligned memory accesses are performed using these byte addresses as shown in Table A3-4 on page A3-8 for the LE endian format and in Table A3-3 on page A3-7 for the BE and BE-32 formats, in each case extended consistently to doubleword accesses. Table H-2 on page AppxH-8 shows which bytes are accessed by each type of aligned memory access and the significance order in which they are accessed.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.3.2 エンディアンのサポート

ARMv4 および ARMv5 では、ビッグエンディアンおよびリトルエンディアンの操作がサポートされます。リトルエンディアンのサポートは、ARMv7 と整合性があります。ビッグエンディアンの制御、構成、および ARM レジスタファイルとメモリとのデータバイトの接続性は異なっています。ただし、相違点は、メモリを使用してビッグエンディアンとリトルエンディアンとのエージェントの間で通信を行う場合にのみ顕在化します。エージェントは、同じプロセッサ上で別のエンディアン形式設定を使用して実行されている、別のプロセッサまたはプログラムです。

ARMv4 および ARMv5 の場合、ビッグエンディアン メモリとリトルエンディアン メモリとの区別は、ワード内のバイトのアドレスを変更することにより管理されます。ARMv7 の場合、ビッグエンディアン メモリとリトルエンディアン メモリとの区別は、バイトアドレスは同じに維持して、ハーフワードまたはワード内のバイトを再配列することにより管理されます。エンディアンのフォーマットは次のとおりです。

LE ARMv4、ARMv5、ARMv6、ARMv7 で使用されるリトルエンディアン フォーマット

BE ARMv6（`SETEND` 命令によりエンディアン形式を制御）および ARMv7 で使用されるビッグエンディアン フォーマット

BE-32 ARMv4、ARMv5、ARMv6（従来のフォーマット。SCTLR.B ビットでエンディアン形式を制御）で使用されるビッグエンディアン フォーマット

BE-32 エンディアンフォーマットでバイトのアドレスを変更する方法を、表 H-1 に示します。この表において、A は ARMv7 メモリマップにおけるダブルワードアラインド アドレスで、S、T、U、V、W、X、Y、Z は A ～ A + 7 のアドレスのバイトです。

表 H-1 各エンディアンフォーマットでのバイトのアドレス

バイト	フォーマット BE または LE でのアドレス	フォーマット BE-32 でのアドレス
S	A	A + 3
T	A + 1	A + 2
U	A + 2	A + 1
V	A + 3	A
W	A + 4	A + 7
X	A + 5	A + 6
Y	A + 6	A + 5
Z	A + 7	A + 4

アラインド メモリアクセスは、LE エンディアンフォーマットの場合は P. A3-8 の表 A3-4 に示されているバイトアドレス、BE および BE-32 フォーマットの場合は P. A3-7 の表 A3-3 に示されているバイトアドレスを使用して実行され、それぞれの場合について整合性のある形でダブルワードアクセスに拡張されます。それぞれのタイプのアラインド メモリアクセスによりアクセスされるバイト、およびアクセスされる順序を P. 付録 H-8 の表 H-2 に示します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table H-2 Bytes accessed by aligned accesses in endian formats

Memory access:		Bytes accessed in endian format:		
Size	**Address**	**LE**	**BE**	**BE-32**
Doubleword	A	ZYXWVUTS	STUVWXYZ	VUTSZYXW
Word	A	VUTS	STUV	VUTS
Word	A+4	ZYXW	WXYZ	ZYXW
Halfword	A	TS	ST	VU
Halfword	A+2	VU	UV	TS
Halfword	A+4	XW	WX	ZY
Halfword	A+6	ZY	YZ	XW
Byte	A	S	S	V
Byte	A+1	T	T	U
Byte	A+2	U	U	T
Byte	A+3	V	V	S
Byte	A+4	W	W	Z
Byte	A+5	X	X	Y
Byte	A+6	Y	Y	X
Byte	A+7	Z	Z	W

Note

If the ARMv4 and ARMv5 endian model was extended to unaligned word and halfword accesses, for example loading a word from byte addresses `0x1001`, `0x1002`, `0x1003`, and `0x1004`, it would not return the same bytes of data to a big endian and little endian agent. However, ARMv4 and ARMv5 do not support unaligned memory access and therefore this cannot occur. In ARMv4 and ARMv5 where use of an unaligned address is permitted, the actual memory access is naturally aligned. See *Alignment* on page AppxH-6.

In ARMv7, all big endian accesses return the same bytes of data from memory as the corresponding little endian accesses. It is only the byte order in the returned value that is different.

For an ARMv4 or ARMv5 implementation, whether the endianness of the memory access is fixed, defined by an input pin on reset, or controlled by the SCTLR.B bit is IMPLEMENTATION DEFINED.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 H-2 各エンディアンフォーマットでアラインドアクセスによりアクセスされるバイト

メモリアクセスの		各エンディアンフォーマットでアクセスされるバイト		
サイズ	**アドレス**	**LE**	**BE**	**BE-32**
ダブルワード	A	ZYXWVUTS	STUVWXYZ	VUTSZYXW
ワード	A	VUTS	STUV	VUTS
ワード	A + 4	ZYXW	WXYZ	ZYXW
ハーフワード	A	TS	ST	VU
ハーフワード	A + 2	VU	UV	TS
ハーフワード	A + 4	XW	WX	ZY
ハーフワード	A + 6	ZY	YZ	XW
バイト	A	S	S	V
バイト	A + 1	T	T	U
バイト	A + 2	U	U	T
バイト	A + 3	V	V	S
バイト	A + 4	W	W	Z
バイト	A + 5	X	X	Y
バイト	A + 6	Y	Y	X
バイト	A + 7	Z	Z	W

注

ARMv4 および ARMv5 のエンディアンモデルが、例えばバイトアドレス `0x1001`、`0x1002`、`0x1003`、`0x1004` からのワードのロードなどのアンアラインドワード アクセスおよびハーフワードアクセスに拡張されている場合は、ビッグエンディアン エージェントとリトルエンディアン エージェントに対してデータの同じバイトは返されません。ただし、ARMv4 および ARMv5 では、アンアラインド メモリアクセスはサポートされていないため、このようなことは発生しません。アンアラインドアドレスの使用が許可されている ARMv4 および ARMv5 の場合、実際のメモリアクセスは自然アラインです。*アライメント* P. 付録 H-6 を参照して下さい。

ARMv7 では、すべてのビッグエンディアン アクセスにおいて、メモリからは対応するリトルエンディアン アクセスと同じデータのバイトが戻されます。返される値のバイト順序のみが異なります。

ARMv4 または ARMv5 実装では、メモリアクセスのエンディアン形式が固定されているか、リセット時に入力ピンによって定義されるか、SCTLR.B ビットによって制御されるかは実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Examples

The distinction between BE and BE-32 is not visible if all agents use the same endian format, because a given memory address always accesses the same location in memory. However, if there are two agents with different endianness the effect is as shown in Example H-1 and Example H-2.

Example H-1 Distinction between BE and BE-32 word stores observed by an LE agent

In this example:

- Agent1 is big endian, `R1=0x1000`, `R2=0x11223344`
- Agent2 is little endian, `R1=0x1000`.

Agent1:

```
STR R2, [R1]
```

Agent2:

```
LDR R2, [R1]          // If Agent1 uses BE-32 endian format: R2 = 0x11223344
                      // If Agent1 uses BE endian format: R2 = 0x44332211
```

Example H-2 Distinction between BE and BE-32 byte stores observed by an LE agent

In this example:

- Agent1 is big endian, `R1=0x1000`, `R2=0x44`, `R3=0x11`
- Agent2 is little endian, `R1=0x1000`.

Agent1:

```
STRB R2, [R1]
STRB R3, [R1, #3]
```

Agent2:

```
LDRB R2, [R1]         // If Agent1 uses BE-32 endian format: R2 = 0x11
                      // If Agent1 uses BE endian format: R2 = 0x44
```

H.3.3 Semaphore support

The only semaphore support in ARMv4 and ARMv5 is provided by the `SWP` and `SWPB` ARM instructions. Use of these instructions is deprecated in ARMv6 and ARMv7 in favour of the exclusive access mechanism provided by `LDREX`, `STREX`, and related instructions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

例

すべてのエージェントが同じエンディアンフォーマットを使用する場合、BE と BE-32 との間の区別は顕在化しません。これは、特定のメモリアドレスは常にメモリの同じ位置にアクセスするためです。ただし、異なるエンディアン形式を持つ 2 つのエージェントが存在する場合、例 H-1 および例 H-2 に示されているような影響が発生します。

例 H-1 LE エージェントにより観察される BE と BE-32 とのワードストアの相違

この例では、次のように想定します。

- Agent1 はビッグエンディアンで、`R1 = 0x1000`、`R2 = 0x11223344` です。
- Agent2 はリトルエンディアンで、`R1 = 0x1000` です。

Agent1：

```
STR R2, [R1]
```

Agent2：

```
LDR R2, [R1]          // If Agent1 uses BE-32 endian format: R2 = 0x11223344
                      // If Agent1 uses BE endian format: R2 = 0x44332211
```

例 H-2 LE エージェントにより観察される BE と BE-32 とのバイトストアの相違

この例では、次のように想定します。

- Agent1 はビッグエンディアンで、`R1 = 0x1000`、`R2 = 0x44`、`R3 = 0x11` です。
- Agent2 はリトルエンディアンで、`R1 = 0x1000` です。

Agent1：

```
STRB R2, [R1]
STRB R3, [R1, #3]
```

Agent2：

```
LDRB R2, [R1]         // If Agent1 uses BE-32 endian format: R2 = 0x11
                      // If Agent1 uses BE endian format: R2 = 0x44
```

H.3.3 セマフォのサポート

ARMv4 および ARMv5 でのセマフォのサポートは、`SWP` および `SWPB` ARM 命令によってのみ提供されます。ARMv6 および ARMv7 では、これらの命令の使用は非推奨です。`LDREX`、`STREX`、および関連する命令によって提供される排他アクセス機構を使用して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.3.4 Memory model and memory ordering

There is no formal definition of the memory model in ARMv4 and ARMv5. ARM implementations generally adopted a Strongly-ordered approach. However the memory order model is IMPLEMENTATION DEFINED.

Memory type support

In ARMv4 and ARMv5 where CP15 is implemented, memory can be tagged using two control bits:

- the B bit (Bufferable), to indicate whether write buffering between the processor and memory is permitted
- the C bit (Cacheable).

Table H-3 shows the ARMv4 and ARMv5 definitions of the C bit and B bit that are interpreted as the formal memory types defined in ARMv6 and ARMv7.

Table H-3 Interpretation of Cacheable and Bufferable bits

C	B	Memory type
0	0	Strongly-ordered
0	1	Device
1	0	Normal, Write-Through Cacheable
1	1	Normal, Write-Back Cacheable

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

H.3.4 メモリモデルとメモリ順序

ARMv4 および ARMv5 では、メモリモデルの正式な定義は存在しません。ARM 実装では、一般にストロングリオーダ手法が採用されています。ただし、メモリオーダ モデルは実装定義です。

メモリタイプのサポート

CP15 が実装されている ARMv4 および ARMv5 の場合、次の 2 つの制御ビットを使用してメモリにタグを付けることができます。

- B ビット（バッファ可能）。プロセッサとメモリとの間でライトバッファが許可されているかどうかを示します。
- C ビット（キャッシュ可能）

ARMv6 および ARMv7 で定義されている正式なメモリタイプとして解釈される、C ビットと B ビットの ARMv4 および ARMv5 での定義を、表 H-3 に示します。

表 H-3 キャッシュ可能ビットおよびバッファ可能ビットの解釈

C	B	メモリタイプ
0	0	ストロングリオーダ
0	1	デバイス
1	0	ノーマル、ライトスルーキャッシュ可能
1	1	ノーマル、ライトバックキャッシュ可能

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.4 Instruction set support

Two instruction sets are supported in ARMv4 and ARMv5:

- the ARM instruction set is supported by all variants of ARMv4 and ARMv5
- the Thumb instruction set is supported by ARMv4T, ARMv5T, ARMv5TE, and ARMv5TEJ.

Floating-point support, identified as VFPv2, was added as an option in ARMv5TE. The VFP instructions are a subset of the coprocessor support in the ARM instruction set, and use coprocessor numbers 10 and 11. The following instructions are not supported in VFPv2, and are specific to ARMv7 VFP support (VFPv3):

- `VMOV` (immediate)
- `VCVT` (between floating-point and fixed-point).

Note

- VFP instruction mnemonics traditionally started with an F. However this has been changed to a V prefix in the Unified Assembler Language introduced in ARMv6T2, and in many cases the rest of the mnemonic has been changed to be more compatible with other instructions mnemonics. This aligns the scalar floating-point support with the ARMv7 Advanced SIMD support, which shares some load/store and move operations to a common register file.
- The VFPv2 instructions are summarized in *F* (former VFP instruction mnemonics)* on page A8-100. This includes the two deprecated instructions in VFPv2 that do not have UAL mnemonics, the `FLDMX` and `FSTMX` instructions.

The instruction sets have grown significantly in ARMv6 and ARMv7 because of:

- the introduction of ARMv6 SIMD
- improved context switching in ARMv6
- the addition of many 32-bit Thumb instructions in ARMv6T2 and ARMv7
- the ThumbEE extension in ARMv7
- addition of the `SMC` instruction with the Security Extensions
- the Advanced SIMD extension in ARMv7.

The ARM and Thumb instruction encodings including the VFP instructions are defined in *Alphabetical list of instructions* on page A8-14.

Note

This appendix describes the instructions included as a mnemonic in ARMv4 and ARMv5. For any mnemonic, to determine which associated instruction encodings appear in a particular architecture variant, see the subsections of *Alphabetical list of instructions* on page A8-14 that describe the mnemonic. Each encoding diagram shows the architecture variants or extensions that include the encoding.

The following sections give more information about ARMv4 and ARMv5 instruction set support:

- *ARM instruction set support* on page AppxH-12
- *Thumb instruction set support* on page AppxH-15
- *System level instruction set support* on page AppxH-17.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.4 命令セットのサポート

ARMv4 および ARMv5 では、次の 2 つの命令セットがサポートされます。

- ARM 命令セットは、ARMv4 および ARMv5 のすべてのバリアントでサポートされます。
- Thumb 命令セットは、ARMv4T、ARMv5T、ARMv5TE、ARMv5TEJ でサポートされます。

浮動小数点サポート VFPv2 が、オプションとして ARMv5TE に追加されています。VFP 命令は ARM 命令セットにおけるコプロセッササポートのサブセットで、コプロセッサ番号 10 および 11 を使用します。次の命令は VFPv2 ではサポートされておらず、ARMv7 の VFP サポート (VFPv3) に固有です。

- `VMOV`（イミディエート）
- `VCVT`（浮動小数点と固定小数点との間）

注

- VFP 命令のニーモニックは従来は F で始まっていましたが、ARMv6T2 で導入された統一アセンブラ言語では V で始まるように変更されました。また、それ以外のニーモニックは多くの場合、他の命令ニーモニックとの互換性が高まるように変更されています。これにより、スカラ浮動小数点サポートと、ARMv7 のアドバンスト SIMD サポートが調整され、一般的なレジスタファイルに対して一部のロード / ストアおよび移動操作が共有されます。
- VFPv2 命令の要約は、*F*（以前の VFP 命令ニーモニック）* P. A8-100 に示されています。ここには、UAL ニーモニックを持たない、VFPv2 では非推奨の `FLDMX` および `FSTMX` 命令が含まれています。

ARMv6 および ARMv7 では、次の理由から命令セットは大幅に拡張されています。

- ARMv6 SIMD の導入
- ARMv6 でコンテキストスイッチが改良されました。
- ARMv6T2 および ARMv7 で多くの 32 ビット Thumb 命令が追加されました。
- ARMv7 で ThumbEE 拡張機能が追加されました。
- セキュリティ拡張機能で `SMC` 命令が追加されました。
- ARMv7 でアドバンスト SIMD 拡張機能が追加されました。

VFP 命令を含む ARM 命令と Thumb 命令のエンコードは、*命令のアルファベット順リスト* P. A8-14 で定義されています。

注

この付録では、ARMv4 および ARMv5 にニーモニックとして含まれている命令について説明します。いずれのニーモニックの場合も、特定のアーキテクチャバリアントに出現する関連命令のエンコードを判別するには、ニーモニックについて記述している*命令のアルファベット順リスト* P. A8-14 のサブセクションを参照して下さい。それぞれのエンコード図には、そのエンコードを含むアーキテクチャバリアントまたは拡張機能が示されています。

次に示すセクションでは、ARMv4 および ARMv5 の命令セットサポートの詳細について示します。

- *ARM 命令セットのサポート* P. 付録 H-12
- *Thumb 命令セットのサポート* P. 付録 H-15
- *システムレベルの命令セットのサポート* P. 付録 H-17

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.4.1 ARM instruction set support

Table H-4 shows the ARM instructions supported in ARMv4 and ARMv5, excluding VFP instructions.

Table H-4 ARM instructions - ARMv4 and ARMv5

Instruction	v4, v4T, v5T	v5TE, v5TEJ
ADC	Yes	Yes
ADD	Yes	Yes
AND	Yes	Yes
B	Yes	Yes
BIC	Yes	Yes
BKPT	v5T only	Yes
BL	Yes	Yes
BLX	v5T only	Yes
BX	v4T and v5T only	Yes
BXJ	No	v5TEJ only
CDP	Yes	Yes
CDP2	v5T only	Yes
CLZ	v5T only	Yes
CMN	Yes	Yes
CMP	Yes	Yes
EOR	Yes	Yes
LDC	Yes	Yes
LDC2	v5T only	Yes
LDM	Yes	Yes
LDR	Yes	Yes
LDRB	Yes	Yes
LDRD	No	Yes

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

H.4.1 ARM 命令セットのサポート

ARMv4 および ARMv5 でサポートされる、VFP 命令以外の ARM 命令を、表 H-4 に示します。

表 H-4 ARM 命令 –ARMv4 および ARMv5

命令	v4、v4T、v5T	v5TE、v5TEJ
ADC	はい	はい
ADD	はい	はい
AND	はい	はい
B	はい	はい
BIC	はい	はい
BKPT	v5T のみ	はい
BL	はい	はい
BLX	v5T のみ	はい
BX	v4T と v5T のみ	はい
BXJ	いいえ	v5TEJ のみ
CDP	はい	はい
CDP2	v5T のみ	はい
CLZ	v5T のみ	はい
CMN	はい	はい
CMP	はい	はい
EOR	はい	はい
LDC	はい	はい
LDC2	v5T のみ	はい
LDM	はい	はい
LDR	はい	はい
LDRB	はい	はい
LDRD	いいえ	はい

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table H-4 ARM instructions - ARMv4 and ARMv5 (continued)

Instruction	v4, v4T, v5T	v5TE, v5TEJ
LDRBT	Yes	Yes
LDRH	Yes	Yes
LDRSB	Yes	Yes
LDRSH	Yes	Yes
LDRT	Yes	Yes
MCR	Yes	Yes
MCR2	v5T only	Yes
MCRR	No	Yes
MLA[a]	Yes	Yes
MOV	Yes	Yes
MRC	Yes	Yes
MRC2	v5T only	Yes
MRRC	No	Yes
MRS	Yes	Yes
MSR	Yes	Yes
MUL[a]	Yes	Yes
MVN	Yes	Yes
ORR	Yes	Yes
PLD	No	Yes
QADD	No	Yes
QDADD	No	Yes
QDSUB	No	Yes
QSUB	No	Yes
RSB	Yes	Yes

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 H-4 ARM 命令 –ARMv4 および ARMv5 （続き）

命令	v4、v4T、v5T	v5TE、v5TEJ
LDRBT	はい	はい
LDRH	はい	はい
LDRSB	はい	はい
LDRSH	はい	はい
LDRT	はい	はい
MCR	はい	はい
MCR2	v5T のみ	はい
MCRR	いいえ	はい
MLA[a]	はい	はい
MOV	はい	はい
MRC	はい	はい
MRC2	v5T のみ	はい
MRRC	いいえ	はい
MRS	はい	はい
MSR	はい	はい
MUL[a]	はい	はい
MVN	はい	はい
ORR	はい	はい
PLD	いいえ	はい
QADD	いいえ	はい
QDADD	いいえ	はい
QDSUB	いいえ	はい
QSUB	いいえ	はい
RSB	はい	はい

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table H-4 ARM instructions - ARMv4 and ARMv5 (continued)

Instruction	v4, v4T, v5T	v5TE, v5TEJ
RSC	Yes	Yes
SBC	Yes	Yes
SMLAL[b]	Yes	Yes
SMLABB, SMLABT, SMLATB, SMLATT	No	Yes
SMLALBB, SMLALBT, SMLALTB, SMLALTT	No	Yes
SMLAWB, SMLAWT	No	Yes
SMULBB, SMULBT, SMULTB, SMULTT	No	Yes
SMULL[b]	Yes	Yes
SMULWB, SMULWT	No	Yes
STC	Yes	Yes
STC2	v5T only	Yes
STM	Yes	Yes
STR	Yes	Yes
STRB	Yes	Yes
STRBT	Yes	Yes
STRD	No	Yes
STRH	Yes	Yes
STRT	Yes	Yes
SUB	Yes	Yes
SVC (previously SWI)	Yes	Yes
SWP	Yes	Yes
SWPB	Yes	Yes
TEQ	Yes	Yes

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 H-4 ARM 命令 –ARMv4 および ARMv5 （続き）

命令	v4、v4T、v5T	v5TE、v5TEJ
RSC	はい	はい
SBC	はい	はい
SMLAL[b]	はい	はい
SMLABB、SMLABT、SMLATB、SMLATT	いいえ	はい
SMLALBB、SMLALBT、SMLALTB、SMLALTT	いいえ	はい
SMLAWB、SMLAWT	いいえ	はい
SMULBB、SMULBT、SMULTB、SMULTT	いいえ	はい
SMULL[b]	はい	はい
SMULWB、SMULWT	いいえ	はい
STC	はい	はい
STC2	v5T のみ	はい
STM	はい	はい
STR	はい	はい
STRB	はい	はい
STRBT	はい	はい
STRD	いいえ	はい
STRH	はい	はい
STRT	はい	はい
SUB	はい	はい
SVC （以前の SWI）	はい	はい
SWP	はい	はい
SWPB	はい	はい
TEQ	はい	はい

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table H-4 ARM instructions - ARMv4 and ARMv5 (continued)

Instruction	v4, v4T, v5T	v5TE, v5TEJ
TST	Yes	Yes
UMLAL[b]	Yes	Yes
UMULL[b]	Yes	Yes

a. The value of APSR.C generated by flag-setting versions of these instructions is UNKNOWN in ARMv4 and is unchanged from ARMv5.
b. The values of APSR.C and APSR.V generated by flag-setting versions of these instructions are UNKNOWN in ARMv4 and are unchanged from ARMv5.

H.4.2 Thumb instruction set support

Table H-5 shows the 16-bit Thumb instructions supported in ARMv4 and ARMv5. ARMv4 before ARMv4T does not support any Thumb instructions.

Table H-5 ARMv4 and ARMv5 support for Thumb instructions

Instruction	v4T	v5T, v5TE, v5TEJ
ADC	Yes	Yes
ADD	Yes	Yes
AND	Yes	Yes
ASR	Yes	Yes
B	Yes	Yes
BIC	Yes	Yes
BKPT	No	Yes
BL	Yes	Yes
BLX	No	Yes
BX	Yes	Yes
CMN	Yes	Yes
CMP	Yes	Yes
EOR	Yes	Yes

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 H-4 ARM 命令 –ARMv4 および ARMv5 （続き）

命令	v4、v4T、v5T	v5TE、v5TEJ
TST	はい	はい
UMLAL[b]	はい	はい
UMULL[b]	はい	はい

a. これらの命令のうち、フラグを設定するバージョンによって生成される APSR.C の値は、ARMv4 では未知で、ARMv5 およびそれ以降では値は変更されません。
b. これらの命令のうち、フラグを設定するバージョンによって生成される APSR.C および APSR.V の値は、ARMv4 では未知で、ARMv5 およびそれ以降では値は変更されません。

H.4.2 Thumb 命令セットのサポート

ARMv4 および ARMv5 でサポートされる 16 ビット Thumb 命令を、表 H-5 に示します。ARMv4T 以前の ARMv4 では、Thumb 命令はサポートされません。

表 H-5 ARMv4 および ARMv5 でサポートされる Thumb 命令

命令	v4T	v5T、v5TE、v5TEJ
ADC	はい	はい
ADD	はい	はい
AND	はい	はい
ASR	はい	はい
B	はい	はい
BIC	はい	はい
BKPT	いいえ	はい
BL	はい	はい
BLX	いいえ	はい
BX	はい	はい
CMN	はい	はい
CMP	はい	はい
EOR	はい	はい

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table H-5 ARMv4 and ARMv5 support for Thumb instructions (continued)

Instruction	v4T	v5T, v5TE, v5TEJ
LDMIA	Yes	Yes
LDR	Yes	Yes
LDRB	Yes	Yes
LDRH	Yes	Yes
LDRSB	Yes	Yes
LDRSH	Yes	Yes
LSL	Yes	Yes
LSR	Yes	Yes
MOV	Yes	Yes
MUL	Yes	Yes
MVN	Yes	Yes
NEG	Yes	Yes
ORR	Yes	Yes
POP	Yes	Yes
PUSH	Yes	Yes
ROR	Yes	Yes
SBC	Yes	Yes
STMIA	Yes	Yes
STR	Yes	Yes
STRB	Yes	Yes
STRH	Yes	Yes
SUB	Yes	Yes
SVC (previously SWI)	Yes	Yes
TST	Yes	Yes

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 H-5 ARMv4 および ARMv5 でサポートされる Thumb 命令 （続き）

命令	v4T	v5T、v5TE、v5TEJ
LDMIA	はい	はい
LDR	はい	はい
LDRB	はい	はい
LDRH	はい	はい
LDRSB	はい	はい
LDRSH	はい	はい
LSL	はい	はい
LSR	はい	はい
MOV	はい	はい
MUL	はい	はい
MVN	はい	はい
NEG	はい	はい
ORR	はい	はい
POP	はい	はい
PUSH	はい	はい
ROR	はい	はい
SBC	はい	はい
STMIA	はい	はい
STR	はい	はい
STRB	はい	はい
STRH	はい	はい
SUB	はい	はい
SVC （以前の SWI）	はい	はい
TST	はい	はい

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.4.3 System level instruction set support

The register and immediate forms of the MRS and MSR instructions are used to manage the CPSR and SPSR as applicable. Other system level instructions are:

- LDM (exception return) and LDM (user registers)
- LDRBT and LDRT
- STM (user registers)
- STRBT and STRT
- SUBS PC, LR and related instructions
- VMRS and VMSR where VFP is supported.

All system level support is from ARM state.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.4.3 システムレベルの命令セットのサポート

レジスタ形式およびイミディエート形式の MRS および MSR 命令は、適用可能な場合は CPSR および SPSR の管理に使用されます。システムレベルの他の命令は次のとおりです。

- LDM（例外からの復帰）および LDM（ユーザレジスタ）
- LDRBT および LDRT
- STM（ユーザレジスタ）
- STRBT および STRT
- SUBS PC, LR および関連する命令
- VMRS および VMSR（VFP がサポートされている場合）

システムレベルのサポートは、すべて ARM 状態から行われます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.5 System level register support

The general registers and programming modes are the same as ARMv7, except that the Security Extensions and Monitor mode are not supported. For more information, see Figure B1-1 on page B1-9. The following sections give information about ARMv4 and ARMv5 system level register support:

- *Program Status Registers (PSRs)*
- *The exception model* on page AppxH-19
- *Execution environment support* on page AppxH-20.

H.5.1 Program Status Registers (PSRs)

The application level programmers' model provides the *Application Program Status Register* (APSR). See *The Application Program Status Register (APSR)* on page A2-14. This is an application level alias for the CPSR. The system level view of the CPSR extends the register, adding state that:

- is used by exceptions
- controls the processor mode.

Each of the exception modes has its own saved copy of the CPSR, the *Saved Program Status Register* (SPSR), as shown in Figure B1-1 on page B1-9. For example, the SPSR for Abort mode is called SPSR_abt.

Note

ARMv4 and ARMv5 do not support Monitor mode and the Security Extensions.

The Current Program Status Register (CPSR)

The CPSR holds the following processor status and control information:

- The APSR, see *APSR support* on page AppxH-4
- The current instruction set state. See *ISETSTATE* on page A2-15, except that:
  - — ThumbEE state is not supported
  - — Jazelle state is supported only in ARMv5TEJ.
- The current processor mode
- Interrupt disable bits.

The non-APSR bits of the CPSR have defined reset values. These are shown in the `TakeReset()` pseudocode function described in *Reset* on page B1-48, except that:

- the CPSR.IT[7:0], CPSR.E and CPSR.A bits are not defined and so do not have reset values
- before ARMv5TEJ, the CPSR.J bit is not defined and so does not have a reset value
- the reset value of CPSR.T is 0.

The rules described in *The Current Program Status Register (CPSR)* on page B1-14 about when mode changes take effect apply with the modification that the ISB can only be the ISB operation described in *c7, Miscellaneous functions* on page AppxH-51.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

H.5 システムレベル レジスタのサポート

セキュリティ拡張機能およびモニタモードがサポートされていないことを除いて、汎用レジスタおよびプログラミングモードは ARMv7 と同じです。詳細については、P. B1-9 の図 B1-1 を参照して下さい。次に示すセクションでは、ARMv4 および ARMv5 におけるシステムレベル レジスタのサポートについて説明します。

- *プログラムステータス レジスタ (PSR)*
- *例外モデル* P. 付録 H-19
- *実行環境のサポート* P. 付録 H-20

H.5.1 プログラムステータス レジスタ (PSR)

アプリケーションレベルのプログラマモデルでは、*アプリケーションプログラム ステータスレジスタ* (APSR) が提供されます。*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14 を参照して下さい。これは、CPSR のアプリケーションレベルのエイリアスです。システムレベルから見た CPSR ではこのレジスタが拡張され、次の状態が追加されます。

- 例外で使用される状態
- プロセッサモードを制御する状態

P. B1-9 の図 B1-1 に示すように、例外モードごとに、CPSR と*保存プログラムステータス レジスタ* (SPSR) の独自の保存コピーが存在します。例えば、アボートモードの SPSR は SPSR_abt という名前です。

注

ARMv4 および ARMv5 では、モニタモードおよびセキュリティ拡張機能はサポートされません。

カレントプログラム ステータスレジスタ (CPSR)

CPSR には、次のプロセッサステータスおよび制御情報が保持されます。

- APSR（*APSR のサポート* P. 付録 H-4 を参照）
- 現在の命令セット状態。*ISETSTATE* P. A2-15 を参照して下さい。ただし、次の点を除きます。
  - – ThumbEE 状態はサポートされません。
  - – Jazelle 状態は ARMv5TEJ でのみサポートされます。
- 現在のプロセッサモード
- 割り込みディセーブルビット

CPSR の非 APSR ビットには、リセット時の値が定義されています。これらは、*リセット* P. B1-48 で説明されている `TakeReset()` 擬似コード関数に示されています。ただし、次の点を除きます。

- CPSR.IT[7:0]、CPSR.E、CPSR.A の各ビットは定義されていないため、リセット時の値はありません。
- ARMv5TEJ 以前では、CPSR.J ビットは定義されていないため、リセット時の値はありません。
- CPSR.T のリセット時の値は 0 です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The Saved Program Status Registers (SPSRs)

The SPSRs are defined as they are in ARMv7, see *The Saved Program Status Registers (SPSRs)* on page B1-15, except that:

- the GE[3:0], IT[7:0], E and A bits are not implemented
- before ARMv5TEJ, the J bit is not implemented.

Format of the CPSR and SPSRs

The format of the CPSR and SPSRs is:

31	30	29	28	27	26	25	24	23 … 8	7	6	5	4 … 0
N	Z	C	V	Q	RAZ /WI		J	Reserved	I	F	T	M[4:0]

The definitions and general rules for the defined PSR bits are the same as ARMv7, see *Format of the CPSR and SPSRs* on page B1-16, except that:

- Before ARMv5TEJ, the J bit is RAZ/WI.
- The T bit of the CPSR, and in ARMv5TEJ the J bit of the CPSR, must not be changed when the CPSR is written by an `MSR` instruction, or else the behavior is UNPREDICTABLE. `MSR` instructions exist only in ARM state in these architecture variants, so this is equivalent to saying the `MSR` instructions in privileged modes must treat these bits as SBZP. `MSR` instructions in User mode still ignore writes to these bits.
- The IT[7:0], GE[3:0], E, and A bitfield definitions for ARMv7 do not apply to ARMv4 and ARMv5.
- Monitor mode is not supported. The associated M[4:0] encoding is a reserved value in ARMv4 and ARMv5.

H.5.2 The exception model

The exception vector offsets and priorities are consistent across all variants of the ARM architecture that use the exception model as stated in *Exceptions* on page B1-30. The Security Extensions and low interrupt latency configuration do not apply to ARMv4 and ARMv5.

In ARMv4 and ARMv5, it is IMPLEMENTATION DEFINED whether high vectors are supported. If they are supported, a hardware configuration input selects whether the normal vectors or the high vectors are used from reset. If high vectors are not supported then SCTLR.V, bit [13], is reserved, RAZ.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

カレントプログラム ステータスレジスタ (CPSR) P. B1-14 で説明されている、モード変更が有効になる時期に関する規則適用されますが、ISB となるのは *c7、その他の機能* P. 付録 H-51 で説明されている ISB 操作のみである、という変更が加えられます。

保存プログラム ステータスレジスタ (SPSR)

SPSR は、ARMv7 と同様に定義されています。*保存プログラム ステータスレジスタ (SPSR)* P. B1-15 を参照して下さい。ただし、次の点を除きます。

- GE[3:0]、IT[7:0]、E、A の各ビットは実装されていません。
- ARMv5TEJ 以前では、J ビットは実装されていません。

CPSR と SPSR のフォーマット

CPSR と SPSR のフォーマットは次のとおりです。

31	30	29	28	27	26	25	24	23 … 8	7	6	5	4 … 0
N	Z	C	V	Q	RAZ/ WI		J	予約	I	F	T	M[4:0]

定義されている PSR ビットの定義と一般的な規則は、ARMv7 と同じです。*CPSR と SPSR のフォーマット* P. B1-16 を参照して下さい。ただし、次の点を除きます。

- ARMv5TEJ 以前では、J ビットは RAZ/WI です。
- CPSR の T ビット（および、ARMv5TEJ では J ビット）は、CPSR に `MSR` 命令を使用して書き込む場合は変更してはいけません。変更した場合、動作は予測不能です。これらのアーキテクチャバリアントでは、`MSR` 命令は ARM 状態でのみ存在するため、これは特権モードで `MSR` 命令がこれらのビットを SBZP として処理する必要があることに相当します。ユーザモードの `MSR` 命令では、これらのビットへの書き込みは無視されます。
- ARMv7 での IT[7:0]、GE[3:0]、E、A の各ビットフィールドの定義は、ARMv4 および ARMv5 には適用されません。
- モニタモードはサポートされません。ARMv4 および ARMv5 では、関連する M[4:0] のエンコードは予約値です。

H.5.2 例外モデル

例外ベクタのオフセットおよび優先度は、*例外* P. B1-30 で説明されているように、例外モデルを使用する ARM アーキテクチャのすべてのバリアント間で一貫しています。セキュリティ拡張機能および低割り込みレイテンシ構成は、ARMv4 および ARMv5 には適用されません。

ARMv4 および ARMv5 では、ハイベクタがサポートされるかどうかは実装定義です。ハイベクタがサポートされる場合、ハードウェア構成入力によって、リセット時から通常ベクタとハイベクタのどちらが使用されるかが選択されます。ハイベクタがサポートされない場合、SCTLR.V（ビット [13]）は予約で、RAZ です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The ARM abort model

ARMv6 and ARMv7 use a *Base Restored Abort Model* (BRAM). However, in ARMv5 and ARMv4 it is IMPLEMENTATION DEFINED whether this model, or a *Base Updated Abort Model* (BUAM) is used. These two abort models are defined as:

Base Restored Abort Model

The base register of any valid load/store instruction that causes a memory system abort is always restored to the value it had immediately before that instruction.

Base Updated Abort Model

After an abort, the base register of any valid load/store instruction that causes a memory system abort is modified by the base register writeback, if any, of that instruction.

The implemented abort model applies uniformly across all instructions.

Exception entry

Entry to exceptions in ARMv4 and ARMv5 is generally as described in the sections:

- *Reset* on page B1-48
- *Undefined Instruction exception* on page B1-49
- *Supervisor Call (SVC) exception* on page B1-52
- *Secure Monitor Call (SMC) exception* on page B1-53
- *Prefetch Abort exception* on page B1-54
- *Data Abort exception* on page B1-55
- *IRQ exception* on page B1-58
- *FIQ exception* on page B1-60.

These ARMv7 descriptions are modified as follows:

- pseudocode statements that set registers, bits and fields that do not exist in the ARMv4 or ARMv5 architecture variant are ignored
- CPSR.T is set to 0, not to SCTLR.TE.

H.5.3 Execution environment support

In ARMv5TEJ, the JOSCR.CV bit is not changed on exception entry in any implementation of Jazelle.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ARM アボートモデル

ARMv6 および ARMv7 では、*ベース復元アボートモデル* (BRAM) が使用されます。ただし、ARMv5 および ARMv4 では、このモデルが使用されるか、または*ベース更新アボートモデル* (BUAM) が使用されるかは実装定義です。これら 2 つのアボートモデルは、次のように定義されます。

ベース復元アボートモデル

メモリシステムのアボートを引き起こす有効なロード / ストア命令のベースレジスタは、常にその命令の直前の値に復元されます。

ベース更新アボートモデル

アボート後、メモリシステムのアボートを引き起こす有効なロード / ストア命令のベースレジスタは、命令のベースレジスタへのライトバック（存在する場合）により変更されます。

実装されているアボートモデルは、すべての命令に同一に適用されます。

例外の開始

ARMv4 および ARMv5 での例外の開始については、次のセクションで一般的な説明があります。

- *リセット* P. B1-48
- *未定義命令例外* P. B1-49
- *スーパバイザコール (SVC) 例外* P. B1-52
- *セキュア モニタコール (SMC) 例外* P. B1-53
- *プリフェッチアボート例外* P. B1-54
- *データアボート例外* P. B1-55
- *IRQ 例外* P. B1-58
- *FIQ 例外* P. B1-60

これらの ARMv7 での説明は、次のように変更されます。

- ARMv4 または ARMv5 アーキテクチャバリアントに存在しないレジスタ、ビット、フィールドを設定する擬似コードステートメントは無視されます。
- CPSR.T は、SCTLR.TE ではなく 0 に設定されます。

H.5.3 実行環境のサポート

ARMv5TEJ では、どの Jazelle 実装においても、例外の開始時に JOSCR.CV ビットは変化しません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.6 System level memory model

The pseudocode listed in *Aligned memory accesses* on page B2-31 and *Unaligned memory accesses* on page B2-32 covers the alignment behavior of all architecture variants from ARMv4. For ARMv4 and ARMv5, SCTLR.U is zero, see *Alignment* on page AppxG-6.

The following sections describe the system level memory model:

- *Cache support*
- *Tightly Coupled Memory (TCM) support*
- *Virtual memory support*
- *Protected memory support* on page AppxH-28.

H.6.1 Cache support

CP15 operations are defined that provide cache operations for managing level 1 instruction, data, or unified caches. Caches can be direct mapped or N-way associative. ARMv4 and ARMv5 define a Cache Type ID Register, to enable software to determine the level 1 cache topology.

ARMv4 and ARMv5 support virtual (virtually indexed, virtually tagged) or physical caches. In a virtual memory system that supports virtual cache or caches, there is no coherence support for virtual aliases that map to the same physical address. When a virtual to physical address mapping changes, caches must be cleaned and invalidated accordingly.

Cache management and flushing of any write buffer in the processor is IMPLEMENTATION DEFINED and managed by CP15. CP15 also supports configuration and control of cache lockdown. For more information on cache management support see *System Control coprocessor (CP15) support* on page AppxH-31, and *c7, Cache operations* on page AppxH-49 and *c9, cache lockdown support* on page AppxH-52.

H.6.2 Tightly Coupled Memory (TCM) support

TCM support in ARMv4 and ARMv5 is IMPLEMENTATION DEFINED.

H.6.3 Virtual memory support

The ARMv4 and ARMv5 translation tables support a similar two level translation table format to the ARMv7 tables. However, there are significant differences in the translation table format because of the following:

- ARMv6 introduced additional bits for encoding memory types, attributes, and extended cache attributes.
- The new translation table format in ARMv6 does not support subpage access permissions.
- ARMv4 does not support 16MB Supersections.
- Only ARMv4 and ARMv5 support tiny (1KB) pages. The fine second level page format is not supported from ARMv6.

For general information about address translation in a VMSA, see *About the VMSA* on page B3-2

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.6 システムレベルのメモリモデル

アライインド メモリアクセス P. B2-31 および *アンアライインドメモリアクセス* P. B2-32 にリストされている擬似コードでは、ARMv4 およびそれ以降のすべてのアーキテクチャバリアントでのアライメント動作が扱われます。ARMv4 および ARMv5 では、SCTLR.U は 0 です。*アライメント* P. 付録 G-6 を参照して下さい。

次に示すセクションでは、システムレベルのメモリモデルについて説明します。

- *キャッシュのサポート*
- *密結合メモリ (TCM) のサポート*
- *仮想メモリのサポート*
- *保護メモリのサポート* P. 付録 H-28

H.6.1 キャッシュのサポート

CP15 操作は、レベル 1 の命令、データ、または統一キャッシュを管理するキャッシュ操作を提供するように定義されます。キャッシュには、直接マップと N ウェイアソシエイティブがあります。ARMv4 および ARMv5 ではキャッシュタイプ ID レジスタが定義されており、ソフトウェアによりレベル 1 キャッシュのトポロジを決定できます。

ARMv4 および ARMv5 では、仮想キャッシュ（仮想インデクス、仮想タグ付き）または物理キャッシュがサポートされます。仮想キャッシュをサポートする仮想メモリシステムでは、同じ物理アドレスにマッピングされる仮想エイリアスのコヒーレンシサポートは存在しません。仮想アドレスから物理アドレスへのマッピングが変更された場合、それに応じてキャッシュをクリーニングして無効化する必要があります。

プロセッサのキャッシュの管理およびライトバッファのフラッシュは実装定義で、CP15 によって管理されます。CP15 では、キャッシュロックダウンの構成と制御もサポートされます。キャッシュ管理のサポートの詳細については、*システム制御コプロセッサ (CP15) のサポート* P. 付録 H-31、および *c7、キャッシュ操作* P. 付録 H-49 と *c9、キャッシュロックダウンのサポート* P. 付録 H-52 を参照して下さい。

H.6.2 密結合メモリ (TCM) のサポート

ARMv4 および ARMv5 での TCM のサポートは実装定義です。

H.6.3 仮想メモリのサポート

ARMv4 および ARMv5 の変換テーブルでは、ARMv7 テーブルに類似した 2 レベルの変換テーブルフォーマットがサポートされます。ただし、次の理由により、変換テーブルフォーマットには大きな相違点が存在します。

- ARMv6 では、メモリタイプ、属性、拡張キャッシュ属性をエンコードするため、追加ビットが導入されています。
- ARMv6 の新規変換テーブルフォーマットでは、サブページのアクセス許可がサポートされていません。
- ARMv4 では、16MB のスーパーセクションがサポートされていません。
- タイニー (1KB) ページは、ARMv4 および ARMv5 でのみサポートされます。第 2 レベルのファイン ページフォーマットは、ARMv6 およびそれ以降ではサポートされていません。

VMSA でのアドレス変換に関する一般的な情報については、*VMSA について* P. B3-2 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The *Fast Context Switch Extension* (FCSE) is an implementation option in ARMv4 and ARMv5 VMSA implementations. For more information, see *FCSE translation* on page B3-4 and Appendix E *Fast Context Switch Extension (FCSE)*.

Note

ARMv7 only supports the new translation table format. ARMv6 supports both old and new formats and uses SCTLR[23] to select which format to use. For more information, see *c1, System Control Register (SCTLR)* on page AppxG-34.

The *Virtual Memory System Architecture* (VMSA) in ARMv4 and ARMv5 supports the following:

- 16MB Supersections, optional support from ARMv5TE
- 1MB Sections
- 64KB Large pages
- 4KB Small pages
- 1KB Tiny pages.

Section virtual to physical address translation is supported by a single level translation table walk. Page address translation requires a two level translation table walk. Each level involves an aligned word read from a translation table in memory with the first level translation table base address held in a CP15 register, TTBR0. Translation table entries are typically cached in a *Translation Lookaside Buffer* (TLB) in the *Memory Management Unit* (MMU) of a given implementation. CP15 operations are used to manage the TLB. For more information, see *c8, VMSA TLB support* on page AppxH-51.

Note

ARMv4 and ARMv5 support a single translation table base address register. TTBR1 and TTBCR were introduced in ARMv6.

Second level translation table accesses are derived from the additional information provided by the first level translation table entry. Two sizes of second level translation table are supported:

- a Coarse page table, where each entry translates a 4KB address space
- a Fine page table, where each entry translates a 1KB address space.

Translation tables are always naturally aligned in memory to the address space they occupy. This means that the least significant n bits of the translation table base address are zero, where $n = \log_2(SIZE)$, and SIZE is the size of the table in bytes.

Translation attributes

ARMv4 and ARMv5 support the following translation table attributes:

- domain access as described in *Domains* on page B3-31
- cacheability with the C and B bits, see *Interpretation of Cacheable and Bufferable bits* on page AppxH-10

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

高速コンテキストスイッチ拡張機能 (FCSE) は、ARMv4 および ARMv5 の VMSA 実装では実装オプションです。詳細については、*FCSE 変換* P. B3-4 と付録 E *「高速コンテキストスイッチ拡張機能 (FCSE)」* を参照して下さい。

注

ARMv7 では、新しい変換テーブルフォーマットのみがサポートされます。ARMv6 では以前のフォーマットと新しいフォーマットの両方がサポートされ、使用するフォーマットは SCTLR[23] で選択します。詳細については、*c1、システム制御レジスタ (SCTLR)* P. 付録 G-34 を参照して下さい。

ARMv4 および ARMv5 の *仮想メモリシステム アーキテクチャ* (VMSA) では、次のものがサポートされます。

- 16MB のスーパーセクション（ARMv5TE およびそれ以降でのサポートはオプション）
- 1MB のセクション
- 64KB のラージページ
- 4KB のスモールページ
- 1KB のタイニーページ

セクションの仮想アドレスから物理アドレスへの変換は、単一レベルの変換テーブルウォークでサポートされます。ページのアドレス変換には、2 レベルの変換テーブルウォークが必要です。各レベルには、CP15 レジスタ (TTBR0) に保持されている第 1 レベルの変換テーブルのベースアドレスを使用した、メモリ内の変換テーブルからのアラインドワード読み出しが含まれます。変換テーブルのエントリは、通常は実装の *メモリ管理ユニット* (MMU) に含まれている *変換ルックアサイドバッファ* (TLB) にキャッシュされます。TLB の管理には、CP15 操作が使用されます。詳細については、*c8、VMSA TLB のサポート* P. 付録 H-51 を参照して下さい。

注

ARMv4 および ARMv5 では、単一の変換テーブルのベースアドレス レジスタがサポートされます。TTBR1 および TTBCR は、ARMv6 で導入されたものです。

第 2 レベルの変換テーブルへのアクセスは、第 1 レベルの変換テーブルのエントリによって提供される追加情報から導き出されます。第 2 レベルの変換テーブルでは、次の 2 つのサイズのテーブルがサポートされます。

- コアース ページテーブル。ここでは、各エントリは 4KB のアドレス空間に変換されます。
- ファイン ページテーブル。ここでは、各エントリは 1KB のアドレス空間に変換されます。

変換テーブルは、占有するアドレス空間に対してメモリ内で常に自然アラインされます。これは、変換テーブルのベースアドレスの最下位 n ビットが 0 であることを意味します。ここで、$n = \log_2(SIZE)$ で、SIZE はバイト単位のテーブルのサイズです。

変換属性

ARMv4 および ARMv5 では、次の変換テーブル属性がサポートされます。

- ドメインアクセス属性（*ドメイン* P. B3-31 を参照）
- C ビットおよび B ビットによるキャッシュ可能属性（*キャッシュ可能ビットおよびバッファ可能ビットの解釈* P. 付録 H-10 を参照）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- access permissions using the AP[1:0], SCTLR.S and SCTLR.R bits as defined in Table H-6 on page AppxH-23
- from ARMv5TE, the option of marking sections as Shareable and support for extended cache attributes using the TEX bitfield with the C and B bits. See Table H-7 and Table H-8 on page AppxH-25.

Table H-6 VMSA access permissions in ARMv4 and ARMv5

SCTLR[a] S	SCTLR[a] R	AP[1:0]	Privileged permissions	User permissions	Description
0	0	00	No access	No access	All accesses generate Permission faults
x	x	01	Read/write	No access	Privileged access only
x	x	10	Read/write	Read-only	Writes in User mode generate Permission faults
x	x	11	Read/write	Read/write	Full access
0	1	00	Read-only	Read-only	Read-only in privileged and User modes
1	0	00	Read-only	No access	Privileged read-only
1	1	00	-	-	Reserved

a. For more information, see *c1, System Control Register (SCTLR)* on page AppxH-39.

Note

Changes to the S and R bits do not affect the access permissions of entries already in the TLB. The TLB must be flushed for the updated S and R bit values to take effect.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- AP[1:0]、SCTLR.S、SCTLR.R の各ビットによるアクセス許可属性（P. 付録 H-23 の表 H-6 で定義）
- セクションを共有可能にマークするオプション、および C ビット、B ビットと TEX ビットフィールドを使用した拡張キャッシュ属性のサポート（ARMv5TE および以降）。表 H-7 および P. 付録 H-25 の表 H-8 を参照して下さい。

表 H-6 ARMv4 および ARMv5 での VMSA アクセス許可

SCTLR[a] S	SCTLR[a] R	AP[1:0]	特権アクセス許可	ユーザアクセス許可	説明
0	0	00	アクセス不可	アクセス不可	すべてのアクセスでアクセス許可フォールトが発生
x	x	01	読み出し / 書き込み	アクセス不可	特権アクセスのみ
x	x	10	読み出し / 書き込み	読み出し専用	ユーザモードで書き込みを行うとアクセス許可フォールトが発生
x	x	11	読み出し / 書き込み	読み出し / 書き込み	完全アクセス
0	1	00	読み出し専用	読み出し専用	特権モードおよびユーザモードにおいて読み出し専用
1	0	00	読み出し専用	アクセス不可	特権読み出し専用
1	1	00	–	–	予約

a. 詳細については、*c1、システム制御レジスタ (SCTLR)* P. 付録 H-39 を参照して下さい。

注

S ビットと R ビットを変更しても、TLB にすでに存在しているエントリのアクセス許可には影響を与えません。更新された S ビットと R ビットの値を有効にするには、TLB をフラッシュする必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

First level descriptor formats

Table H-7 shows the translation table first level descriptor formats:

- Supersection support, the TEX field, and the S bit are only permitted from ARMv5TE. Where these features are not supported, the corresponding bits must be zero.
- Supersections can support address translation from a 32-bit virtual address to a physical address of up to 40 bits.

Table H-7 ARMv4 and ARMv5 first level descriptor format

	31 … 24	23 … 20	19	18	17	16	15	14 … 12	11	10	9	8 … 5	4	3	2	1	0
Fault	Ignore															0	0
Coarse page table	Coarse page table base address										P	Domain		SBZ		0	1
Section	Section base address		SBZ	0	SBZ	S[a]	SBZ	TEX[b]		AP	P	Domain	SBZ	C	B	1	0
Supersection	Supersection base address	PA[35:32] optional	SBZ	1	SBZ	S	SBZ	TEX		AP	P	PA[39:36] optional	SBZ	C	B	1	0
Fine page table	Fine page table base address									SBZ	P	Domain		SBZ		1	1

a. S=1 indicates Shareable memory. For more information, see *Summary of ARMv7 memory attributes* on page A3-25.
b. From ARMv5TE, the TEX bits can be used with the C and B bits as described in *C, B, and TEX[2:0] encodings without TEX remap* on page B3-33.

Bits [1:0] of the descriptor identify the descriptor type:

0b00 Invalid or fault entry.

0b01 Coarse page table descriptor. Bits [31:10] of the descriptor give the physical address of a second level translation table.

0b10 Section or Supersection descriptor for the associated Modified Virtual Address (MVA). Bits [31:20] of the descriptor give the Section address, bits [31:24] provide the Supersection address. Bit [18] indicates which to use when both are supported.

0b11 Fine page table descriptor. Bits [31:12] of the descriptor give the physical address of a second level translation table.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 1 レベル記述子のフォーマット

変換テーブルの第 1 レベル記述子のフォーマットを、表 H-7 に示します。

- スーパーセクションのサポート、TEX フィールド、S ビットは、ARMv5TE およびそれ以降でのみ許可されます。これらの機能がサポートされていない場合、対応するビットは 0 にする必要があります。
- スーパーセクションでは、32 ビットの仮想アドレスから最大 40 ビットの物理アドレスへのアドレス変換をサポートできます。

表 H-7 ARMv4 および ARMv5 での第 1 レベル記述子のフォーマット

	31 … 24	23 … 20	19	18	17	16	15	14 … 12	11	10	9	8 … 5	4	3	2	1	0
フォールト	無視															0	0
コアース ページ テーブル	コアース ページテーブルのベースアドレス										P	ドメイン		SBZ		0	1
セクション	セクションのベースアドレス		SBZ	0	SBZ	S[a]	SBZ	TEX[b]		AP	P	ドメイン	SBZ	C	B	1	0
スーパー セクション	スーパーセクション ベースアドレス	PA[35:32] (オプション)	SBZ	1	SBZ	S	SBZ	TEX		AP	P	PA[39:36] (オプション)	SBZ	C	B	1	0
ファイン ページ テーブル	ファイン ページテーブルのベースアドレス									SBZ	P	ドメイン		SBZ		1	1

a. S = 1 は、共有可能メモリを示します。詳細については、*ARMv7 のメモリ属性の要約* P. A3-25 を参照して下さい。
b. ARMv5TE およびそれ以降では、*TEX 再マップを伴わない C、B、TEX[2:0] のエンコード* P. B3-33 で説明されているように、TEX ビットは C ビットおよび B ビットとともに使用できます。

記述子のビット [1:0] により、次のように記述子のタイプが識別されます。

0b00 無効または不正なエントリ。

0b01 コアース ページテーブル記述子。記述子のビット [31:10] で、第 2 レベルの変換テーブルの物理アドレスを指定します。

0b10 関連付けられている修飾仮想アドレス (MVA) のセクションまたはスーパーセクション記述子。記述子のビット [31:20] でセクションアドレスを、ビット [31:24] でスーパーセクションアドレスを指定します。両方のセクションがサポートされている場合は、ビット [18] で使用するセクションの種類を指定します。

0b11 ファイン ページテーブル記述子。記述子のビット [31:12] で、第 2 レベルの変換テーブルの物理アドレスを指定します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Second level Coarse page table descriptor format

In a Coarse page table, each entry provides translation information for 4KB of memory. Large page table entries must be repeated 16 times to ensure translation of all addresses in the page. Tiny pages are not supported. Coarse page tables are 1KB in size and must be aligned on a 1KB boundary.

Table H-8 shows the translation table second level descriptor formats for a second level Coarse page table.

Table H-8 Second level Coarse page descriptor formats

31 … 16	15	14 … 12	11	10	9	8	7	6	5	4	3	2	1	0
Fault	Ignore												0	0
Large page	Large page base address	SBZ	TEX			AP3		AP2		AP1		AP0	C	B	0	1
Small page	Small page base address					AP3		AP2		AP1		AP0	C	B	1	0
Extended small page	Extended small page base address optional in ARMv5TE, otherwise reserved					SBZ			TEX			AP	C	B	1	1

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

第 2 レベルのコアース ページテーブル記述子のフォーマット

コアース ページテーブルでは、各エントリにより 4KB のメモリに関する変換情報が提供されます。ラージ ページテーブルのエントリは、ページのすべてのアドレスが変換されることを保証するため、16 回繰り返す必要があります。タイニーページはサポートされません。コアース ページテーブルのサイズは 1KB で、1KB 境界にアラインする必要があります。

第 2 レベルのコアース ページテーブルに使用される、変換テーブルの第 2 レベル記述子のフォーマットを、表 H-8 に示します。

表 H-8 第 2 レベルのコアースページ記述子のフォーマット

31 … 16	15	14 … 12	11	10	9	8	7	6	5	4	3	2	1	0
フォールト	無視												0	0
ラージページ	ラージページのベースアドレス	SBZ	TEX			AP3		AP2		AP1		AP0	C	B	0	1
スモールページ	スモールページのベースアドレス					AP3		AP2		AP1		AP0	C	B	1	0
拡張スモールページ	拡張スモールページのベースアドレス（ARMv5TE ではオプション、その他の場合は予約）					SBZ			TEX			AP	C	B	1	1

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Second level Fine page table descriptor format

In a Fine page table, each entry provides translation information for 1KB of memory. Large page table entries must be repeated 64 times and Small page entries four times to ensure translation of all addresses. Fine tables are 4KB in size and must be aligned on a 4KB boundary.

Table H-9 shows the translation table second level descriptor format for a second level Fine page table.

Table H-9 Second level Fine page table descriptor format

31 … 16	15 … 12	11	10	9	8	7	6	5	4	3	2	1	0
Fault	Ignore											0	0
Large page	Large page base address		SBZ	AP3	AP2	AP1	AP0			C	B	0	1
Small page	Small page base address			AP3	AP2	AP1	AP0			C	B	1	0
Tiny page	Tiny page base address				SBZ			AP		C	B	1	1

Bits [1:0] of the descriptor identify the descriptor type:

0b00 Invalid or fault entry. The associated MVA is unmapped, and attempting to access it generates a Translation fault.

0b01 Large page descriptor. Bits [31:16] of the descriptor give the base address of the Large page.

0b10 Small page descriptor. Bits [31:12] of the descriptor give the base address of the Small page.

0b11 Tiny page descriptor. Bits [31:10] of the descriptor give the base address of the Tiny page.

Subpage support

In the Large page and Small page formats in Table H-9, the page is divided into four equal sized subpages. The AP0, AP1, AP2, and AP3 fields are the AP bits for subpages 0, 1, 2, and 3 respectively. Subpage 0, controlled by the AP0 field, is the subpage with the lowest base address.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

第 2 レベルのファイン ページテーブル記述子のフォーマット

ファイン ページテーブルでは、各エントリにより 1KB のメモリに関する変換情報が提供されます。すべてのアドレスの変換を保証するため、ラージ ページテーブルのエントリは 64 回、スモール ページのエントリは 4 回繰り返す必要があります。ファインテーブルのサイズは 4KB で、4KB 境界にアラインする必要があります。

第 2 レベルのファイン ページテーブルに使用される、変換テーブルの第 2 レベル記述子のフォーマットを、表 H-9 に示します。

表 H-9 第 2 レベルのファイン ページテーブル記述子のフォーマット

31 … 16	15 … 12	11	10	9	8	7	6	5	4	3	2	1	0
フォールト	無視											0	0
ラージ ページ	ラージページのベースアドレス		SBZ	AP3	AP2	AP1	AP0			C	B	0	1
スモール ページ	スモールページのベースアドレス			AP3	AP2	AP1	AP0			C	B	1	0
タイニー ページ	タイニーページのベースアドレス				SBZ			AP		C	B	1	1

記述子のビット [1:0] により、次のように記述子のタイプが識別されます。

0b00 無効または不正なエントリ。関連する MVA がマッピングされていないため、その MVA にアクセスを試みると変換フォールトが発生します。

0b01 ラージページ記述子。記述子のビット [31:16] で、ラージページのベースアドレスを指定します。

0b10 スモールページ記述子。記述子のビット [31:12] で、スモールページのベースアドレスを指定します。

0b11 タイニーページ記述子。記述子のビット [31:10] で、タイニーページのベースアドレスを指定します。

サブページのサポート

表 H-9 のラージページおよびスモールページのフォーマットでは、ページは 4 つの等しいサイズのサブページに分割されます。AP0、AP1、AP2、AP3 フィールドは、それぞれサブページ 0、1、2、3 の AP ビットです。サブページ 0 は AP0 フィールドによって制御され、ベースアドレスが最も小さいサブページです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Translation table walks

An MVA and TTBR0 are used to access translation table information as follows:

- For a Section translation. See Figure B3-4 on page B3-17 with N == 0.
- For a Large page translation using a Coarse page table access. See Figure B3-7 on page B3-20 with N == 0.
- For a Small page translation using a Coarse page table access. See Figure B3-6 on page B3-19 with N == 0.
- For a Tiny page translation using a Fine page table access. See Figure H-1.

Note

A Large page table or Small page table translation is performed on a Fine page table access by reducing the second level page table base address to bits [31:12] and extending the second level table index to MVA[19:10].

Figure H-1 Tiny page address translation, VMSAv5 and VMSAv4 only

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

変換テーブルウォーク

MVA および TTBR0 は、変換テーブル情報にアクセスするため次の方法で使用されます。

- セクション変換。P. B3-17 の図 B3-4 を参照して下さい（ただし、N == 0）。
- コアース ページテーブルへのアクセスを使用したラージページ変換。P. B3-20 の図 B3-7 を参照して下さい（ただし、N == 0）。
- コアース ページテーブルへのアクセスを使用したスモールページ変換。P. B3-19 の図 B3-6 を参照して下さい（ただし、N == 0）。
- ファイン ページテーブルへのアクセスを使用したタイニーページ変換。図 H-1 を参照して下さい。

注

ラージ ページテーブル変換またはスモール ページテーブル変換は、第 2 レベルのページテーブルのベースアドレスをビット [31:12] に縮小し、第 2 レベルのテーブルインデクスを MVA[19:10] に拡張することにより、ファイン ページテーブルへのアクセス時に実行されます。

図 H-1 タイニーページのアドレス変換（VMSAv5 と VMSAv4 のみ）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.6.4 Protected memory support

The MPU based *Protected Memory System Architecture* (PMSA) is a much simpler memory protection scheme than the MMU-based VMSA model described in *Virtual memory support* on page AppxH-21. The simplification applies to both the hardware and the software. PMSA in ARMv4 and ARMv5 differs from that supported in ARMv6 and ARMv7 in the following ways:

- the programming model is unique to ARMv4 and ARMv5
- the supported number of memory regions is fixed
- background memory support requires use of a region resource
- there is no architecturally-defined recovery mechanism from memory aborts
- there is no default memory map definition.

Control and configuration

CP15 registers are used to fully define *protection regions*, eliminating the VMSA requirements for hardware to do translation table walks, and for software to set up and maintain the translation tables. This makes memory checking fully deterministic. However, the level of control is now region based rather than page based. This means the control is not as fine-grained.

The following features apply:

- The memory is divided into regions. CP15 registers are used to define the region size, base address, and memory attributes. For example, cacheability, bufferability, and access permissions of a region.
- Memory region control (read and write access) is permitted only from privileged modes.
- If an address is defined in multiple regions, a fixed priority scheme (highest region number) is used to define the properties of the address being accessed.
- An access to an address that is not defined in any region causes a memory abort.
- All addresses are physical addresses. Address translation is not supported.
- PMSA supports unified (von Neumann) and separate (Harvard) instruction and data address spaces.

Eight regions can be configured, with C, B, and AP[1:0] attribute bits associated with each region. The supported region sizes are 2NKB, where 2 =< N =< 32. It is IMPLEMENTATION DEFINED if the regions are configurable or fixed in an implementation:

- as eight unified regions supporting data accesses and instruction fetches
- as eight data regions and eight instruction regions each with independent memory region attributes.

CP15 provides the following support:

- a global MPU enable bit, SCTLR.M
- cacheability register support, a C bit for each region
- bufferability register support, a B bit for each region

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

H.6.4 保護メモリのサポート

MPU ベースの*保護メモリシステム アーキテクチャ*(PMSA) は、*仮想メモリのサポート* P. 付録 H-21 で説明されている MMU ベースの VMSA モデルより大幅に単純なメモリ保護方式です。この単純化は、ハードウェアとソフトウェアの両方に当てはまります。ARMv4 および ARMv5 での PMSA は、ARMv6 および ARMv7 でサポートされている PMSA とは次の点が異なります。

- プログラミングモデルは ARMv4 および ARMv5 に固有です。
- サポートされるメモリ領域の数は固定です。
- バックグラウンドメモリのサポートには、領域リソースを使用することが必要です。
- メモリアボートから回復するための機構は、アーキテクチャで定義されていません。
- デフォルトのメモリマップ定義は存在しません。

制御と構成

CP15 レジスタは、ハードウェアによる変換テーブルウォークの実行と、ソフトウェアによる変換テーブルのセットアップと保守を行うための VMSA 要件を不要にするため、*保護領域*を完全に定義します。これにより、メモリのチェックが完全に予測可能になります。ただし、制御のレベルはページベースではなく領域ベースになります。これは、制御の粒度が粗くなることを意味します。

次の機能が適用されます。

- メモリは領域に分割されます。CP15 レジスタは、領域サイズ、ベースアドレス、メモリ属性の定義に使用されます。例えば、キャッシュ可能属性、バッファ可能属性、領域へのアクセス許可などです。
- メモリ領域の制御（読み出し / 書き込みアクセス）は、特権モードでのみ許可されます。
- 複数の領域でアドレスが定義されている場合は、アクセスされるアドレスのプロパティを定義するために固定優先度方式（最大の領域番号）が使用されます。
- どの領域でも定義されていないアドレスにアクセスすると、メモリアボートが発生します。
- アドレスはすべて物理アドレスです。アドレス変換はサポートされません。
- PMSA では、統一の (フォン ノイマン)、および独立した (ハーバード) 命令とデータのアドレス空間がサポートされます。

8 つの領域を構成でき、各領域に対して C、B、AP[1:0] の属性ビットが割り当てられます。サポートされる領域サイズは 2NKB です（ここで、2 =< N =< 32）。領域は次のいずれかで、どちらかに固定されているか、構成が可能かは実装定義です。

- データアクセスおよび命令フェッチをサポートする 8 つの統一領域
- それぞれ独立したメモリ領域属性を持つ 8 つのデータ領域と 8 つの命令領域

CP15 では、次の設定がサポートされます。

- グローバルな MPU イネーブルビット、SCTLR.M
- キャッシュ可能属性レジスタのサポート、各領域の C ビット
- バッファ可能属性レジスタのサポート、各領域の B ビット
- AP7[1:0] ～ AP0[1:0] の 2 ビットの許可フィールドを提供するアクセス許可レジスタのサポート、各領域の AP ビットフィールド
- 4 ビットの AP フィールドの拡張アクセス許可レジスタのサポート（オプション）
- 各領域のベースアドレス、サイズフィールド、イネーブルビットを提供する領域レジスタ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- access permission register support that provides AP7[1:0] to AP0[1:0] 2-bit permission fields, an AP bitfield for each region
- optional extended access permission register support for 4-bit AP fields
- region registers providing a base address, size field, and an enable bit for each region.

For details of the PMSA support in CP15 see *c2, c3, c5, and c6, PMSA support* on page AppxH-43.

The C and B bits are configured according to the type of memory that is to be accessed. For more information, see *Memory type support* on page AppxH-10. Table H-10 defines the standard AP bit behavior.

Table H-10 PMSA access permissions in ARMv4 and ARMv5

AP[1:0]	Privileged permissions	User permissions	Description
00	No access	No access	All accesses generate Permission faults
01	Read/write	No access	Privileged access only
10	Read/write	Read-only	Writes in User mode generate Permission faults
11	Read/write	Read/write	Full access

Some implementations also include support for read-only access permission. Table H-11 defines the extended AP bit behavior.

Table H-11 PMSA extended access permissions in ARMv4 and ARMv5

AP[3:0]	Privileged permissions	User permissions	Description
0000	No access	No access	All accesses generate a Permission fault
0001	Read/write	No access	Privileged access only
0010	Read/write	Read-only	Writes in User mode generate a Permission fault
0011	Read/write	Read/write	Full access
0100	UNPREDICTABLE	UNPREDICTABLE	-
0101	Read-only	No access	Privileged read-only access
0110	Read-only	Read-only	Read-only access
0111	UNPREDICTABLE	UNPREDICTABLE	-
1xxx	UNPREDICTABLE	UNPREDICTABLE	-

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

CP15 での PMSA サポートの詳細については、*c2*、*c3*、*c5*、*および c6*、*PMSA のサポート* P. 付録 H-43 を参照して下さい。

C ビットと B ビットは、アクセスするメモリのタイプに応じて構成されます。詳細については、*メモリタイプのサポート* P. 付録 H-10 を参照して下さい。標準 AP ビットの動作の定義を、表 H-10 に示します。

表 H-10 ARMv4 および ARMv5 での PMSA アクセス許可

AP[1:0]	特権アクセス許可	ユーザアクセス許可	説明
00	アクセス不可	アクセス不可	すべてのアクセスでアクセス許可フォールトが発生
01	読み出し / 書き込み	アクセス不可	特権アクセスのみ
10	読み出し / 書き込み	読み出し専用	ユーザモードで書き込みを行うとアクセス許可フォールトが発生
11	読み出し / 書き込み	読み出し / 書き込み	完全アクセス

一部の実装には、読み出し専用アクセス許可のサポートも含まれています。拡張 AP ビットの動作の定義を、表 H-11 に示します。

表 H-11 ARMv4 および ARMv5 での PMSA 拡張アクセス許可

AP[3:0]	特権アクセス許可	ユーザアクセス許可	説明
0000	アクセス不可	アクセス不可	すべてのアクセスでアクセス許可フォールトが発生
0001	読み出し / 書き込み	アクセス不可	特権アクセスのみ
0010	読み出し / 書き込み	読み出し専用	ユーザモードで書き込みを行うと、アクセス許可フォールトが発生
0011	読み出し / 書き込み	読み出し / 書き込み	完全アクセス
0100	予測不能	予測不能	–
0101	読み出し専用	アクセス不可	特権読み出し専用アクセス
0110	読み出し専用	読み出し専用	読み出し専用アクセス
0111	予測不能	予測不能	–
1xxx	予測不能	予測不能	–

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Memory access sequence

When the ARM processor generates a memory access, the MPU compares the memory address with the programmed memory regions as follows:

- If a matching memory region is not found, a memory abort is signaled to the processor.
- If a matching memory region is found, the region information is used as follows:
  - The access permission bits are used to determine whether the access is permitted. If the access is not permitted, the MPU signals a memory abort. Otherwise, the access can proceed.
  - The memory region attributes are used to determine the access attributes, for example cached or non-cached, as described in *Memory type support* on page AppxH-10.

Note

When a Permission fault occurs, there is no fault status information provision for PMSA in ARMv4 or ARMv5. The CP15 registers FSR and FAR are only available in implementations with VMSA support.

Overlapping regions

The Protection Unit can be programmed with two or more overlapping regions. When overlapping regions are programmed, a fixed priority scheme is applied to determine the region whose attributes are applied to the memory access.

Attributes for region 7 take highest priority and those for region 0 take lowest priority. For example:

- Data region 2 is programmed to be 4KB in size, starting from address `0x3000` with AP == 0b010 (privileged modes full access, User mode read-only).
- Data region 1 is programmed to be 16KB in size, starting from address `0x0` with AP == 0b001 (privileged mode access only).

When the processor performs a data load from address `0x3010` while in User mode, the address falls into both region 1 and region 2. Because there is a clash, the attributes associated with region 2 are applied. In this case, the load would not abort.

Background region

Overlapping regions increase the flexibility of how regions can be mapped onto physical memory devices in the system. The overlapping properties can also be used to specify a background region. For example, assume a number of physical memory areas sparsely distributed across the 4GB address space. If only these regions are configured, any access outside the defined sparse address space aborts. You can override this behavior by programming region 0 to be a 4GB background region. In this case, if the address does not fall into any of the other regions, the access is controlled by the attributes specified for region 0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

メモリアクセスのシーケンス

ARM プロセッサでメモリアクセスが発生すると、MPU により、メモリアドレスとプログラムされているメモリ領域とが次のように比較されます。

- 一致するメモリ領域が見つからない場合は、プロセッサにメモリアボートが通知されます。
- 一致するメモリ領域が見つかった場合は、その領域情報が次のように使用されます。
  - アクセス許可ビットを使用して、アクセスが許可されているかどうかが判断されます。アクセスが許可されていない場合は、MPU によりメモリアボートが通知されます。それ以外の場合は、アクセスを実行することができます。
  - *メモリタイプのサポート* P. 付録 H-10 で説明されているように、メモリ領域属性を使用して、アクセス属性（例えば、キャッシュ対象または非キャッシュ）が判別されます。

注

ARMv4 または ARMv5 では、アクセス許可フォールトが発生した場合、PMSA のフォールトステータス情報の機構は存在しません。CP15 レジスタ FSR および FAR は、VMSA をサポートする実装でのみ使用可能です。

オーバラップしている領域

保護ユニットには、2 つ以上のオーバラップしている領域をプログラムできます。オーバラップしている領域がプログラムされると、固定優先度方式により、どの領域の属性をメモリアクセスに適用するかが決定されます。

領域 7 の属性が最も優先度が高く、領域 0 の属性は優先度が最低です。次に例を示します。

- データ領域 2 は、アドレス `0x3000` から始まる 4KB のサイズにプログラムされ、AP == 0b010（特権モードで完全アクセス、ユーザモードで読み出し専用）です。
- データ領域 1 は、アドレス `0x0` から始まる 16KB のサイズにプログラムされ、AP == 0b001（特権モードアクセスのみ）です。

プロセッサがユーザモードでアドレス `0x3010` からデータのロードを実行すると、そのアドレスは領域 1 と領域 2 の両方に存在します。競合するため、領域 2 に関連付けられた属性が適用されます。この場合、ロードはアボートしません。

バックグラウンド領域

領域のオーバラップにより、領域をシステムの物理メモリデバイスにより柔軟にマッピングできます。オーバラップしているプロパティは、バックグラウンド領域の指定にも使用できます。例えば、多数の物理メモリ領域が 4GB のアドレス空間に間隔をあけて分散していると仮定します。これらの領域のみが構成されている場合、定義されている一部のアドレス空間に含まれないアドレスへのアクセスはアボートします。この動作は、領域 0 を 4GB のバックグラウンド領域としてプログラムすることによりオーバライドできます。この場合、アドレスが他の領域のいずれにも該当しなければ、アクセスは領域 0 用に指定された属性により制御されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.7 System Control coprocessor (CP15) support

Before ARMv6, it is IMPLEMENTATION DEFINED whether a System Control coprocessor, CP15, is implemented. However, support of ID registers, control registers, cache support, and memory management with virtual or protected memory support resulted in the widespread adoption of a standard for control and configuration of these features. That standard is described here. With the exception of a small number of operations and supporting registers, for example the memory barrier operations described in *c7, Miscellaneous functions* on page AppxH-51, all CP15 accesses require privileged access.

The following sections summarize the CP15 registers known to have been supported in ARMv4 or ARMv5 implementations:

- *Organization of CP15 registers in an ARMv4 or ARMv5 VMSA implementation* on page AppxH-32
- *Organization of CP15 registers in an ARMv4 or ARMv5 PMSA implementation* on page AppxH-33.

For details of the registers provided by a particular implementation see the appropriate Technical Reference Manual, or other product documentation.

The rest of this section describes the ARMv4 and ARMv5 CP15 support in order of the CRn value.

Note

Definitions of CP15 registers in this appendix apply to both VMSA and PMSA implementations unless otherwise indicated.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.7 システム制御コプロセッサ（CP15）のサポート

ARMv6 以前では、システム制御コプロセッサ CP15 が実装されているかどうかは実装定義です。ただし、ID レジスタと制御レジスタ、キャッシュのサポート、および仮想メモリサポートまたは保護メモリサポートを使用したメモリ管理をサポートするため、これらの機能の制御と構成を行うための標準が広く採用されています。この標準については本章で説明されています。少数の操作およびそれをサポートするレジスタ、例えば *c7、その他の機能* P. 付録 H-51 で説明されているメモリバリア操作などの例外を除いて、すべての CP15 アクセスでは特権アクセスが必要です。

次に示すセクションで、ARMv4 または ARMv5 実装でサポートされている CP15 レジスタの要約について説明します。

- *ARMv4 または ARMv5 の VMSA 実装での CP15 レジスタの編成* P. 付録 H-32
- *ARMv4 または ARMv5 の PMSA 実装での CP15 レジスタの編成* P. 付録 H-33

特定の実装で提供されるレジスタの詳細については、該当するテクニカルリファレンス マニュアルまたは他の製品説明書を参照して下さい。

このセクションの以下の部分では、ARMv4 および ARMv5 での CP15 サポートについて、CRn 値の順に説明します。

注

特に指定がない場合、この付録に記載されている CP15 レジスタの定義は、VMSA および PMSA 実装の両方に適用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.7.1 Organization of CP15 registers in an ARMv4 or ARMv5 VMSA implementation

Figure H-2 shows the CP15 registers in an ARMv4 or ARMv5 VMSA implementation:

Figure H-2 CP15 registers in a VMSAv4 or VMSAv5 implementation

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.7.1 ARMv4 または ARMv5 の VMSA 実装での CP15 レジスタの編成

ARMv4 または ARMv5 の VMSA 実装での CP15 レジスタを、図 H-2 に示します。

図 H-2 VMSAv4 または VMSAv5 実装での CP15 レジスタ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.7.2 Organization of CP15 registers in an ARMv4 or ARMv5 PMSA implementation

Figure H-3 shows the CP15 registers in an ARMv4 or ARMv5 PMSA implementation:

Figure H-3 CP15 registers in a PMSAv4 or PMSAv5 implementation

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.7.2 ARMv4 または ARMv5 の PMSA 実装での CP15 レジスタの編成

ARMv4 または ARMv5 の PMSA 実装での CP15 レジスタを、図 H-3 に示します。

図 H-3 PMSAv4 または PMSAv5 実装での CP15 レジスタ

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.7.3 c0, ID support

ARMv4 and ARMv5 implementations support the following ID registers:

- *Main ID Register* (MIDR). See *c0, Main ID Register (MIDR)*.
- *Cache Type Register* (CTR). See *c0, Cache Type Register (CTR)* on page AppxH-35.
- Optionally, the *TCM Type Register* (TCMTR). See *c0, TCM Type Register (TCMTR)* on page AppxH-38.

Table H-12 shows how these read-only registers are accessed using the `MRC` instruction.

Table H-12 ID register support

Register	CRn	opc1	CRm	opc2
Main ID Register, MIDR	c0	0	c0	0
Cache Type ID Register, CTR,	c0	0	c0	1
TCM Type Register, TCMTR	c0	0	c0	2
Aliases of MIDR	c0	0	c0	3, 4, 5, 6, 7

c0, Main ID Register (MIDR)

This register is as described for ARMv7 if either:

- the implementer code in MIDR bits [31:24] is not `0x41`
- the top four bits of the primary part number in MIDR bits [15:4] are neither `0x0` nor `0x7`.

If the implementer code is `0x41` and the top four bits of the primary part number are `0x0`, the processor is an obsolete ARMv2 or ARMv3 processor.

If the implementer code is `0x41` and the top four bits of the primary part number are `0x7`, then:

- If bit[23] is 0, the processor is an obsolete ARMv3 processor.
- If bit[23] is 1, the processor is an ARMv4T processor and bits[22:16] are an IMPLEMENTATION DEFINED variant number. Bits[31:24,15:0] are as described for ARMv7.

For the ARMv7 descriptions of the MIDR see:

- *c0, Main ID Register (MIDR)* on page B3-81 for a VMSA implementation
- *c0, Main ID Register (MIDR)* on page B4-32 for a PMSA implementation.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

H.7.3 c0、ID サポート

ARMv4 および ARMv5 実装では、次の ID レジスタがサポートされます。

- *メイン ID レジスタ*(MIDR)。*c0、メイン ID レジスタ (MIDR)*を参照して下さい。
- キャッシュタイプ レジスタ(CTR)。*c0、キャッシュタイプ レジスタ (CTR)* P. 付録 H-35 を参照して下さい。
- *TCM タイプレジスタ*(TCMTR)（オプション）。*c0、TCM タイプレジスタ (TCMTR)* P. 付録 H-38 を参照して下さい。

これらの読み出し専用レジスタに、`MRC` 命令を使用してアクセスする方法を、表 H-12 に示します。

表 H-12 ID レジスタのサポート

レジスタ	CRn	opc1	CRm	opc2
メイン ID レジスタ、MIDR	c0	0	c0	0
キャッシュタイプ ID レジスタ、CTR	c0	0	c0	1
TCM タイプレジスタ、TCMTR	c0	0	c0	2
MIDR のエイリアス	c0	0	c0	3、4、5、6、7

c0、メイン ID レジスタ (MIDR)

次のいずれかの場合、このレジスタは ARMv7 について説明されているものと同じです。

- MIDR のビット [31:24] の実装者コードが `0x41` でない
- MIDR のビット [15:4] の主要部品番号の上位 4 ビットが、`0x0` と `0x7` のいずれでもない

実装者コードが `0x41` で、主要部品番号の上位 4 ビットが `0x0` の場合、プロセッサは廃止された ARMv2 または ARMv3 プロセッサです。

実装者コードが `0x41` で、主要部品番号の上位 4 ビットが `0x7` の場合、次のようになります。

- ビット [23] が 0 の場合、プロセッサは廃止された ARMv3 プロセッサです。
- ビット [23] が 1 の場合、プロセッサは ARMv4T プロセッサで、ビット [22:16] は実装定義のバリアント番号です。ビット [31:24、15:0] は、ARMv7 についての説明と同じです。

ARMv7 での MIDR の説明については、次のセクションを参照して下さい。

- *c0、メイン ID レジスタ (MIDR)* P. B3-81（VMSA 実装の場合）
- *c0、メイン ID レジスタ (MIDR)* P. B4-32 （PMSA 実装の場合）

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c0, Cache Type Register (CTR)

The format of the Cache Type Register is significantly different from the ARMv7 definition described in *c0, Cache Type Register (CTR)* on page B3-83. However, the general properties described by the register, and the access rights for the register, are unchanged.

This section describes the implementation of the CP15 c0 Cache Type Register and is applicable to a VMSA or PMSA implementation.

The Cache Type Register supplies the following details about the level 1 cache implementation:

- whether there is a unified cache or separate instruction and data caches
- the cache size, line length, and associativity
- whether it is a Write-Through cache or a Write-Back cache
- the cache cleaning and lockdown capabilities.

The format of the Cache Type Register is:

31 29	28 25	24	23 12	11 0
0 0 0	Ctype	S	DSize	ISize

Ctype, bits [28:25]

Cache type field. Specifies details of the cache not indicated by the S bit and the Dsize and Isize fields. Table H-13 shows the encoding of this field. All values not specified in the table are reserved.

S, bit [24] Separate caches bit. The meaning of this bit is:

0 Unified cache

1 Separate instruction and data caches.

If S == 0, the Isize and Dsize fields both describe the unified cache, and must be identical.

Dsize, bits [23:12]

Specifies the size, line length and associativity of the data cache, or of the unified cache if S == 0. For details of the encoding see *Cache size fields* on page AppxH-36.

Isize, bits [11:0]

Specifies the size, line length and associativity of the instruction cache, or of the unified cache if S == 0. For details of the encoding see *Cache size fields* on page AppxH-36.

Table H-13 shows the Ctype values that can be used in the CTR:

Table H-13 Cache type values

Ctype[a]	Cache method	Cache lockdown[b]
0b0000	Write-Through	Not supported
0b0010	Write-Back	Not supported

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c0、キャッシュタイプ レジスタ (CTR)

キャッシュタイプ レジスタのフォーマットは、*c0、キャッシュタイプ レジスタ (CTR)* P. B3-83 で説明されている ARMv7 での定義とは大きく異なります。ただし、レジスタによって記述される汎用プロパティ、およびレジスタのアクセス権限は変更されていません。

このセクションでは、CP15 c0 キャッシュタイプ レジスタの実装について説明します。内容は VMSA または PMSA 実装に適用することができます。

キャッシュタイプ レジスタは、レベル 1 のキャッシュ実装に関する次の情報を提供します。

- 統一キャッシュと、独立した命令およびデータキャッシュとのどちらが存在するか
- キャッシュのサイズ、ライン長、アソシエティビティ
- ライトスルーキャッシュであるかライトバックキャッシュであるか
- キャッシュのクリーニングとロックダウン機能

キャッシュタイプ レジスタのフォーマットは次のとおりです。

31 29	28 25	24	23 12	11 0
0 0 0	Ctype	S	DSize	ISize

Ctype、ビット [28:25]

キャッシュタイプ フィールド。S ビットおよび Dsize と Isize フィールドで指定されていないキャッシュの詳細を指定します。このフィールドのエンコードを表 H-13 に示します。この表で指定されていない値はすべて予約されています。

S、ビット [24] 独立キャッシュビット。このビットの意味は次のとおりです。

0 統一キャッシュ

1 独立した命令キャッシュとデータキャッシュ

S == 0 の場合、Isize および Dsize フィールドはともに統一キャッシュについて記述しているため、同一の必要があります。

Dsize、ビット [23:12]

データキャッシュ、または統一キャッシュ（S == 0 の場合）のサイズ、ライン長、アソシエティビティを指定します。エンコードの詳細については、*キャッシュサイズ フィールド* P. 付録 H-36 を参照して下さい。

Isize、ビット [11:0]

命令キャッシュ、または統一キャッシュ（S == 0 の場合）のサイズ、ライン長、アソシエティビティを指定します。エンコードの詳細については、*キャッシュサイズ フィールド* P. 付録 H-36 を参照して下さい。

CTR で使用できる Ctype の値を、表 H-13 に示します。

表 H-13 キャッシュタイプの値

Ctype[a]	キャッシュ方式	キャッシュロックダウン[b]
0b0000	ライトスルー	未サポート
0b0010	ライトバック	未サポート

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table H-13 Cache type values (continued)

Ctype[a]	Cache method	Cache lockdown[b]
0b0101	Write-Back	Format D
0b0110[c]	Write-Back	Format A
0b0111[c]	Write-Back	Format B
0b1110	Write-Back	Format C

a. CType values not shown are reserved and must not be used.
b. For details see *c9, cache lockdown support* on page AppxH-52.
c. In ARMv6 this Ctype value is reserved and must not be used.

For details of the CP15 c7 operations used for cleaning Write-Back caches see *c7, Cache operations* on page AppxH-49.

Cache size fields

The Dsize and Isize fields in the CTR have the same format:

23 11	22 10	21 9	18 6	17 5	15 3	14 2	13 1	12 0
P	0	Size		Assoc		M	Len	

P For a VMSA implementation, indicates whether the allocation of bits [13:12] of the virtual address is restricted, imposing the *page coloring* restriction. The meaning of this field is:
- **0** No restriction, or PMSA implementation
- **1** Page coloring restriction applies, see *Virtual to physical translation mapping restrictions* on page AppxG-26.

Size Indicates the size of the cache, but is qualified by the M bit, see Table H-14 on page AppxH-37.

Assoc Indicates the associativity of the cache, but is qualified by the M bit, see *Cache associativity* on page AppxH-37.

M Qualifies the values in the Size and Assoc subfields.

Len Specifies the line length of the cache. The possible values of this field are:
- **0b00** Line length is 2 words (8 bytes)
- **0b01** Line length is 4 words (16 bytes)
- **0b10** Line length is 8 words (32 bytes)
- **0b11** Line length is 16 words (64 bytes).

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 H-13 キャッシュタイプの値 （続き）

Ctype[a]	キャッシュ方式	キャッシュロックダウン[b]
0b0101	ライトバック	フォーマット D
0b0110[c]	ライトバック	フォーマット A
0b0111[c]	ライトバック	フォーマット B
0b1110	ライトバック	フォーマット C

a. ここに示されていない CType の値は予約されており、使用できません。
b. 詳細については、*c9、キャッシュロックダウンのサポート* P. 付録 H-52 を参照して下さい。
c. ARMv6 ではこの Ctype の値は予約されており、使用できません。

ライトバック キャッシュのクリーニングに使用される CP15 c7 操作の詳細については、*c7、キャッシュ操作* P. 付録 H-49 を参照して下さい。

キャッシュサイズ フィールド

CTR の Dsize および Isize フィールドは同じフォーマットです。

23 11	22 10	21 9	18 6	17 5	15 3	14 2	13 1	12 0
P	0	Size		Assoc		M	Len	

P VMSA 実装の場合、仮想アドレスのビット [13:12] の割り当てが制限されるかどうか、つまり *ページのカラーリング*制限を示します。このフィールドの意味は次のとおりです。
- 0 制限なし、または PMSA 実装
- 1 ページのカラーリング制限が適用されます。*仮想アドレスから物理アドレスへの変換マッピングに関する制約条件* P. 付録 G-26 を参照して下さい。

Size キャッシュのサイズを示します。ただし、M ビットにより修飾されます。P. 付録 H-37 の表 H-14 を参照して下さい。

Assoc キャッシュのアソシエティビティを示します。ただし、M ビットにより修飾されます。*キャッシュのアソシエティビティ* P. 付録 H-37 を参照して下さい。

M Size および Assoc サブフィールドの値を修飾します。

Len キャッシュの長さを指定します。このフィールドの可能な値は次のとおりです。
- 0b00 ライン長は 2 ワード（8 バイト）です。
- 0b01 ライン長は 4 ワード（16 バイト）です。
- 0b10 ライン長は 8 ワード（32 バイト）です。
- 0b11 ライン長は 16 ワード（64 バイト）です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table H-14 shows how the size of the cache is determined by the Size field and M bit.

Table H-14 Cache sizes

Size field	Size if M == 0	Size if M == 1
0b0000	0.5KB	0.75KB
0b0001	1KB	1.5KB
0b0010	2KB	3KB
0b0011	4KB	6KB
0b0100	8KB	12KB
0b0101	16KB	24KB
0b0110	32KB	48KB
0b0111	64KB	96KB
0b1000	128KB	192KB

Cache associativity

Table H-15 show how the associativity of the cache is determined by the Assoc field and the M bit.

Table H-15 Cache associativity

Assoc field	Associativity if:	
	M == 0	M == 1
0b000	1 way (direct mapped)	Cache absent
0b001	2 way	3 way
0b010	4 way	6 way
0b011	8 way	12 way
0b100	16 way	24 way
0b101	32 way	48 way
0b110	64 way	96 way
0b111	128 way	192 way

The Cache absent encoding overrides all other data in the cache size field.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Size フィールドおよび M ビットによるキャッシュのサイズの決定方法を、表 H-14 に示します。

表 H-14 キャッシュサイズ

Size フィールド	M = 0 の場合のサイズ	M = 1 の場合のサイズ
0b0000	0.5KB	0.75KB
0b0001	1KB	1.5KB
0b0010	2KB	3KB
0b0011	4KB	6KB
0b0100	8KB	12KB
0b0101	16KB	24KB
0b0110	32KB	48KB
0b0111	64KB	96KB
0b1000	128KB	192KB

キャッシュのアソシエティビティ

Assoc フィールドおよび M ビットによるキャッシュのアソシエティビティの決定方法を、表 H-15 に示します。

表 H-15 キャッシュのアソシエティビティ

Assoc フィールド	次の場合のアソシエティビティ	
	M = 0	M = 1
0b000	1 ウェイ（直接マップ）	キャッシュなし
0b001	2 ウェイ	3 ウェイ
0b010	4 ウェイ	6 ウェイ
0b011	8 ウェイ	12 ウェイ
0b100	16 ウェイ	24 ウェイ
0b101	32 ウェイ	48 ウェイ
0b110	64 ウェイ	96 ウェイ
0b111	128 ウェイ	192 ウェイ

キャッシュなしのエンコードは、キャッシュサイズ フィールドの他のすべてのデータに優先します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Excluding the cache absent case (Assoc == 0b000, M == 1) you can use the following formulae to determine the values LINELEN, ASSOCIATIVITY, and NSETS (number of sets) from the Size, Assoc and Len fields of the CTR. These formulae give the associativity values shown in Table H-15 on page AppxH-37:

```
LINELEN        = 1 << (Len+3)                /* In bytes */
MULTIPLIER     = 2 + M
NSETS          = 1 << (Size + 6 - Assoc - Len)
ASSOCIATIVITY  = MULTIPLIER << (Assoc - 1)
```

Multiplying these together gives the overall cache size as:

```
CACHE_SIZE     = MULTIPLIER << (Size+8)      /* In bytes */
```

Note

Cache length fields with (Size + 6 - Assoc - Len) < 0 are invalid, because they correspond to impossible combinations of line length, associativity, and overall cache size. So the formula for NSETS never involves a negative shift value.

c0, TCM Type Register (TCMTR)

In an ARMv4 or ARMv5 implementation that supports CP15 and TCM, the TCMTR is an optional register. For details of the TCMTR implementation see *c0, TCM Type Register (TCMTR)* on page AppxG-33.

H.7.4 c1, System control register support

ARMv4 and ARMv5 implementations support the following system control registers:

- a System Control Register (SCTLR)
- an IMPLEMENTATION DEFINED Auxiliary Control Register (ACTLR).

Table H-16 shows how the registers are accessed using the `MCR` and `MRC` instructions.

Table H-16 System control register support

Register	CRn	opc1	CRm	opc2
System Control Register, SCTLR	c1	0	c0	0
Auxiliary Control Register, ACTLR	c1	0	c0	1

SCTLR is the primary system configuration register in CP15.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

キャッシュが存在しない場合 (Assoc == 0b000、M == 1) 以外では、CTR の Size、Assoc、Len の各フィールドに基づいて次の式を使用し、LINELEN、ASSOCIATIVITY、NSETS（セットの数）を決定できます。これらの式から、P. 付録 H-37 の表 H-15 に示されるアソシエティビティの値を求めることができます。

```
LINELEN        = 1 << (Len+3)                /* In bytes */
MULTIPLIER     = 2 + M
NSETS          = 1 << (Size + 6 - Assoc - Len)
ASSOCIATIVITY  = MULTIPLIER << (Assoc - 1)
```

これらの値を乗算すると、次のようにキャッシュ全体のサイズを求めることができます。

```
CACHE_SIZE     = MULTIPLIER << (Size+8)      /* In bytes */
```

注

(Size + 6 − Assoc − Len) < 0 であるキャッシュ長フィールドは無効です。これらの値に相当する、ライン長、アソシエティビティ、キャッシュ全体のサイズの組み合わが存在しないためです。このため、NSETS を求める式に負のシフト値が含まれることはありません。

c0、TCM タイプレジスタ (TCMTR)

CP15 および TCM をサポートする ARMv4 または ARMv5 実装では、TCMTR はオプションのレジスタです。TCMTR 実装の詳細については、*c0、TCM タイプレジスタ (TCMTR)* P. 付録 G-33 を参照して下さい。

H.7.4 c1、システム制御レジスタのサポート

ARMv4 および ARMv5 実装では、次のシステム制御レジスタがサポートされます。

- システム制御レジスタ (SCTLR)
- 実装定義の補助制御レジスタ (ACTLR)

`MCR` および `MRC` 命令を使用してこれらのレジスタにアクセスする方法を、表 H-16 に示します。

表 H-16 システム制御レジスタのサポート

レジスタ	CRn	opc1	CRm	opc2
システム制御レジスタ、SCTLR	c1	0	c0	0
補助制御レジスタ、ACTLR	c1	0	c0	1

SCTLR は、CP15 の 1 次システム構成レジスタです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c1, System Control Register (SCTLR)

This section describes the implementation of the System Control Register, SCTLR, for ARMv4 and ARMv5. The format of the SCTLR is:

31 … 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	L4	RR	V	I	Z	F	R	S	B	L	D	P	W	C	A	M

Bits [31:16] Reserved.

These reserved bits in the SCTLR are allocated in some circumstances:

- bits [19:16] have been associated with TCM support
- bit [26], described as the L2 bit, has been used to indicate level 2 cache support, see *Level 2 cache support* on page AppxH-51.

These usage models are not compatible with ARMv7.

L4, Bit [15] This bit inhibits ARMv5T Thumb interworking behavior when set. It stops bit [0] updating the CPSR.T bit. Use of the feature is deprecated in ARMv6 and the feature is not supported in ARMv7.

RR, bit [14] Round Robin bit. This bit selects an alternative replacement strategy with a more easily predictable worst-case performance if the cache implementation supports this functionality:

0 Normal replacement strategy, for example random replacement

1 Predictable strategy, for example round robin replacement.

The replacement strategy associated with each value of the RR bit is IMPLEMENTATION DEFINED.

V, bit [13] Vectors bit. This bit selects the base address of the exception vectors:

0 Normal exception vectors, base address `0x00000000`

1 High exception vectors (Hivecs), base address `0xFFFF0000`.
This base address is never remapped.

Support of the V bit is IMPLEMENTATION DEFINED. An implementation can include a configuration input signal that determines the reset value of the V bit. If there is no configuration input signal to determine the reset value of this bit, it resets to 0.

I, bit [12] Instruction cache enable bit. This is a global enable bit for instruction caches:

0 Instruction caches disabled

1 Instruction caches enabled.

If the system does not implement any instruction caches that can be accessed by the processor at any level of the memory hierarchy, this bit is RAZ/WI.

If the system implements any instruction caches that can be accessed by the processor then it must be possible to disable them by setting this bit to 0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c1、システム制御レジスタ (SCTLR)

このセクションでは、ARMv4 および ARMv5 におけるシステム制御レジスタ (SCTLR) の実装について説明します。SCTLR のフォーマットは次のとおりです。

31 … 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
予約	L4	RR	V	I	Z	F	R	S	B	L	D	P	W	C	A	M

ビット[31:16] 予約。

SCTLR のこれらの予約ビットは、一部の環境では次のように割り当てられます。

- ビット [19:16] は、TCM サポートに関連付けられています。
- L2 ビットとして説明されているビット [26] は、レベル 2 のキャッシュサポートを示すために使用されています。*レベル 2 キャッシュのサポート* P. 付録 H-51 を参照して下さい。

これらの使用モデルは、ARMv7 と互換性はありません。

L4、ビット[15] このビットがセットされている場合、ARMv5T の Thumb インターワーキング動作が禁止されます。また、ビット [0] による CPSR.T ビットの更新が停止されます。この機能の使用は ARMv6 では非推奨で、ARMv7 ではこの機能がサポートされていません。

RR、ビット[14] ラウンドロビン ビット。キャッシュ実装でこの機能がサポートされている場合、このビットにより、ワーストケースのパフォーマンスの予測を容易にするため、次のような代替置換方式が選択されます。

0 ランダム置換などの通常の置換方式

1 ラウンドロビン置換などの予測可能な置換方式

RR ビットのそれぞれの値に関連付けられている置換方式は、実装定義です。

V、ビット[13] ベクタビット。このビットにより、次のように例外ベクタのベースアドレスが選択されます。

0 通常の例外ベクタ、ベースアドレス `0x00000000`

1 上位例外ベクタ (Hivecs)、ベースアドレス `0xFFFF0000`
このベースアドレスは再マッピングされることはありません。

V ビットのサポートは実装定義です。実装には、V ビットのリセット時の値を決定する構成入力信号を含めることができます。リセット時の値を決定する構成入力信号がない場合、このビットのリセット時の値は 0 です。

I、ビット[12] 命令キャッシュ イネーブルビット。命令キャッシュのグローバルなイネーブルビットです。

0 命令キャッシュは非稼動です。

1 命令キャッシュは稼動しています。

システムのメモリ階層のどのレベルにも、プロセッサからアクセス可能な命令キャッシュが実装されていない場合、このビットは RAZ/WI です。

システムにプロセッサからアクセス可能な命令キャッシュが実装されている場合は、このビットを 0 にセットしてそれらのキャッシュを非稼動にできる必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Z, bit [11] Branch prediction enable bit. This bit is used to enable branch prediction, also called program flow prediction:

0 program flow prediction disabled

1 program flow prediction enabled.

If program flow prediction cannot be disabled, this bit is RAO/WI. Program flow prediction includes all possible forms of speculative change of instruction stream prediction. Examples include static prediction, dynamic prediction, and return stacks.

If the implementation does not support program flow prediction this bit is RAZ/WI.

F (bit [10]) The meaning of this bit is IMPLEMENTATION DEFINED.

R (bit [9]) ROM protection bit, supported for backwards compatibility. The effect of this bit is described in Table H-6 on page AppxH-23. Use of this feature is deprecated in ARMv6 and the feature is not supported in ARMv7.

S (bit [8]) System protection bit, supported for backwards compatibility. The effect of this bit is described in Table H-6 on page AppxH-23. Use of this feature is deprecated in ARMv6 and the feature is not supported in ARMv7.

B (bit [7]) This bit configures the ARM processor to the endianness of the memory system:

0 Little-endian memory system (LE)

1 Big-endian memory system (BE-32).

ARM processors that support both little-endian and big-endian memory systems use this bit to configure the ARM processor to rename the four byte addresses in a 32-bit word.

Endian support changed in ARMv6. Use of this feature is deprecated in ARMv6 and the feature is not supported in ARMv7.

An implementation can include a configuration input signal that determines the reset value of the B bit. If there is no configuration input signal to determine the reset value of this bit then it resets to 0.

Bits [6:4] RAO/SBOP.

W (bit [3]) This is the enable bit for the write buffer:

0 Write buffer disabled

1 Write buffer enabled.

If the write buffer is not implemented, this bit is RAZ/WI. If the write buffer cannot be disabled, this bit is RAO and ignores writes. Use of this feature is deprecated in ARMv6 and the feature is not supported in ARMv7

C, bit [2] Cache enable bit. This is a global enable bit for data and unified caches:

0 Data and unified caches disabled

1 Data and unified caches enabled.

If the system does not implement any data or unified caches that can be accessed by the processor at any level of the memory hierarchy, this bit is RAZ/WI.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

Z、ビット[11] 分岐予測イネーブルビット。このビットは分岐予測（プログラムフロー予測とも呼ばれます）を可能にするために使用されます。

0 プログラムフロー予測は不可能です。

1 プログラムフロー予測が可能です。

プログラムフロー予測を不可能にできない場合、このビットは RAO/WI です。プログラムフロー予測には、命令ストリーム予測の投機的変更の使用可能な形式がすべて含まれます。例としては、静的予測、動的予測、リターンスタックなどがあります。

実装でプログラムフロー予測がサポートされていない場合、このビットは RAZ/WI です。

F、ビット[10] このビットの意味は実装定義です。

R、ビット[9] ROM 保護ビット。下位互換性のためサポートされています。このビットの影響を、P. 付録 H-23 の表 H-6 に示します。この機能の使用は ARMv6 では非推奨で、ARMv7 ではこの機能はサポートされていません。

S、ビット[8] システム保護ビット。下位互換性のためサポートされています。このビットの影響を、P. 付録 H-23 の表 H-6 に示します。この機能の使用は ARMv6 では非推奨で、ARMv7 ではこの機能はサポートされていません。

B、ビット[7] このビットは、ARM プロセッサのメモリシステムのエンディアン形式を構成します。

0 リトルエンディアンのメモリシステム (LE)

1 ビッグエンディアンのメモリシステム (BE-32)

リトルエンディアンおよびビッグエンディアンの両方のメモリシステムがサポートされる ARM プロセッサでは、このビットを使用して 32 ビットワードの 4 バイトアドレスの名前が変更されます。

ARMv6 では、エンディアンのサポートが変更されています。この機能の使用は ARMv6 では非推奨で、ARMv7 ではこの機能はサポートされていません。

実装には、B ビットのリセット時の値を決定する構成入力信号を含めることができます。リセット時の値を決定する構成入力信号がない場合、このビットのリセット時の値は 0 です。

ビット [6:4] RAO/SBOP。

W、ビット[3] これは、ライトバッファのイネーブルビットです。

0 ライトバッファは非稼動です。

1 ライトバッファは稼動中です。

ライトバッファが実装されていない場合、このビットは RAZ/WI です。ライトバッファを非稼動にできない場合、このビットは RAO で、書き込みは無視されます。この機能の使用は ARMv6 では非推奨で、ARMv7 ではこの機能はサポートされていません。

C、ビット[2] キャッシュイネーブル ビット。データキャッシュおよび統一キャッシュのグローバルなイネーブルビットです。

0 データキャッシュおよび統一キャッシュは非稼動です。

1 データキャッシュおよび統一キャッシュが稼動中です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If the system implements any data or unified caches that can be accessed by the processor then it must be possible to disable them by setting this bit to 0.

A, bit [1] Alignment bit. This is the enable bit for Alignment fault checking:

0 Alignment fault checking disabled

1 Alignment fault checking enabled.

For more information, see *Alignment* on page AppxH-6.

M, bit [0] Memory control bit. This is a global enable bit to enable an MMU where VMSA is supported, or an MPU where PMSA is supported:

0 memory management (MMU or MPU) disabled

1 memory management (MMU or MPU) enabled.

H.7.5 c2 and c3, VMSA memory protection and control registers

ARMv4 and ARMv5 support a single Translation Table Base Register (TTBR) that is compatible with TTBR0, and the *Domain Access Control Register* (DACR).

The TTBR is as defined for TTBR0 in *CP15 c2, Translation table support registers* on page B3-113 except that:

- The base address bitfield is a fixed-length field, bits [31:14] (N=0)
- Bit [5] is reserved.

The DACR is as defined in *c3, Domain Access Control Register (DACR)* on page B3-119.

H.7.6 c5 and c6, VMSA memory system support

ARMv4 and ARMv5 support a Fault Status Register (FSR) and a Fault Address Register (FAR). These registers are accessed using `MCR` and `MRC` instructions. Table H-17 summarizes them.

Table H-17 VMSA fault support

Register	CRn	opc1	CRm	opc2
Fault Status Register, FSR	c5	0	c0	0
Fault Address Register, FAR	c6	0	c0	0

The FSR is updated on Prefetch Abort exceptions and Data Abort exceptions. The FAR is only updated with the MVA on Data Abort exceptions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

システムのメモリ階層のどのレベルにも、プロセッサからアクセス可能なデータまたは統一キャッシュが実装されていない場合、このビットは RAZ/WI です。

システムに、プロセッサからアクセス可能なデータまたは統一キャッシュが実装されている場合は、このビットを 0 にセットして、それらのキャッシュを非稼動にできる必要があります。

A、ビット[1] アライメントビット。アライメントフォールト チェックのイネーブルビットです。

0 アライメントフォールト チェックが不可能です。

1 アライメントフォールト チェックが可能です。

詳細については、*アライメント* P. 付録 H-6 を参照して下さい。

M、ビット[0] メモリ制御ビット。VMSA がサポートされている場合は MMU を、PMSA がサポートされている場合は MPU を稼動するためのグローバルなイネーブルビットです。

0 メモリ管理（MMU または MPU）が非稼動です。

1 メモリ管理（MMU または MPU）が稼動中です。

H.7.5 c2 および c3、VMSA メモリ保護および制御レジスタ

ARMv4 および ARMv5 では、TTBR0 と互換性のある単一の変換テーブル ベースレジスタ (TTBR)、および*ドメインアクセス制御レジスタ*(DACR) がサポートされます。

TTBR は、*CP15 c2、変換テーブルサポート レジスタ* P. B3-113 で説明されている TTBR0 の場合と同様に定義されますが、次の例外があります。

- ベースアドレス ビットフィールドは固定長のフィールド（ビット [31:14] (N = 0)）です。
- ビット [5] は予約されています。

DACR は、*c3、ドメインアクセス制御レジスタ (DACR)* P. B3-119 で説明されているように定義されます。

H.7.6 c5 および c6、VMSA メモリシステムのサポート

ARMv4 および ARMv5 では、フォールト ステータス レジスタ (FSR) およびフォールトアドレス レジスタ (FAR) がサポートされます。これらのレジスタには、`MCR` および `MRC` 命令を使用してアクセスします。要約を表 H-17 に示します。

表 H-17 VMSA フォールトのサポート

レジスタ	CRn	opc1	CRm	opc2
フォールト ステータス レジスタ、FSR	c5	0	c0	0
フォールトアドレス レジスタ、FAR	c6	0	c0	0

FSR は、プリフェッチアボート例外およびデータアボート例外の際に更新されます。FAR は、データアボート例外の際の MVA でのみ更新されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In ARMv5 and ARMv4 implementations the format of the FSR is:

31	10	9	8	7	4	3	0
Reserved	a			Domain		FS[3:0]	

a. It is IMPLEMENTATION DEFINED whether bit [10] is reserved or supports an additional fault status bit, FS[4].

Bits [31:11, 9:8]

Reserved, UNK/SBZP.

Bit [10] FS[4] where defined, otherwise UNK/SBZP.

Domain, bits [7:4]

The domain of the fault address.

FS, bits [3:0] Fault status bits. Indicate the cause of the fault.

Table H-18 lists the base level of fault status encodings returned in the FSR

Table H-18 VMSAv5 and VMSAv4 FSR encodings

FSR[10]	FSR[3:0]	Source of fault		Domain
0	00x1	Alignment fault		Invalid
0	0101 0111	Translation fault	Section Page	Invalid Valid
0	1001 1011	Domain fault	Section Page	Valid Valid
0	1100 1110	Translation table walk External Abort	1st level 2nd level	Invalid Valid
0	1101 1111	Permission fault	Section Page	Valid Valid
0	0xx0 10x0	IMPLEMENTATION DEFINED[a]		-
1	xxxx	IMPLEMENTATION DEFINED[a]		-

a. ARM recommends that any additional codes are compatible with those defined for ARMv6 and ARMv7 as described in Table B3-11 on page B3-50 and Table B3-12 on page B3-51.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

ARMv5 および ARMv4 実装では、FSR のフォーマットは次のとおりです。

31	10	9	8	7	4	3	0
予約	a			ドメイン		FS[3:0]	

a. ビット [10] が予約されているか、または追加のフォールトステータス ビット (FS[4]) をサポートするかは実装定義です。

ビット [31:11、9:8]

予約。UNK/SBZP。

ビット [10] FS[4]（定義されている場合）、UNK/SBZP（定義されていない場合）。

ドメイン、ビット [7:4]

フォールトアドレスのドメイン。

FS、ビット [3:0] フォールトステータス ビット。フォールトの原因を示します。

FSR で戻されるフォールトステータス エンコードのベースレベルの一覧を、表 H-18 に示します。

表 H-18 VMSAv5 および VMSAv4 での FSR のエンコード

FSR[10]	FSR[3:0]	フォールトのソース		ドメイン
0	00x1	アライメントフォールト		無効
0	0101 0111	変換フォールト	セクション ページ	無効 有効
0	1001 1011	ドメインフォールト	セクション ページ	有効 有効
0	1100 1110	変換テーブルウォーク外部アボート	第 1 レベル 第 2 レベル	無効 有効
0	1101 1111	アクセス許可フォールト	セクション ページ	有効 有効
0	0xx0 10x0	実装定義[a]		–
1	xxxx	実装定義[a]		–

a. P. B3-50 の表 B3-11 および P. B3-51 の表 B3-12 で説明されているように、追加コードはすべて、ARMv6 および ARMv7 で定義されているコードと互換性を持たせるようにすることをお勧めします。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.7.7 c2, c3, c5, and c6, PMSA support

While the general principles for memory protection in ARMv4 and ARMv5 are the same, CP15 support for protected memory is different from the programming model of ARMv6 and ARMv7. Memory regions have configurable base address and size attributes. There are also registers for describing cacheability, bufferability, and access permissions across the regions. For more information, see *Memory model and memory ordering* on page AppxH-10.

ARMv4 and ARMv5 support a fixed number of memory regions, either:

- eight unified memory regions
- eight data and eight instruction regions.

Table H-19 shows the PMSA register support.

Table H-19 PMSA register support

Register	CRn	opc1	CRm	opc2
Data or unified Cacheability Register, DCR	c2	0	c0	0
Instruction Cacheability Register, ICR	c2	0	c0	1
Data or unified Bufferability Register, DBR	c3	0	c0	0
Data or unified Access Permission Register, DAPR	c5	0	c0	0
Instruction Access Permission Register, IAPR	c5	0	c0	1
Data or unified Extended Access Permission Register, DEAPR	c5	0	c0	2
Instruction Extended Access Permission Register, IEAPR	c5	0	c0	3
Data or unified Memory Region Registers, DMRR0-DMRR7	c6	0	c0-c7[a]	0
Instruction Memory Region Registers, IMRR0-IMRR7	c6	0	c0-c7[a]	1

a. `<CRm>` selects the region, for example `<CRm>` == 6 selects the region register for region 6, DMRR6 or IMRR6.

If an implementation has a single set of protection regions that apply to both instruction and data accesses, only the registers that are accessed using even values of `<opc2>` exist. Where separate data and instruction regions are supported, with the exception of the extended access permission registers, registers associated with data have `<opc2>` == 0 and those associated with instructions have `<opc2>` == 1. All PMSA registers are 32-bit registers and only accessible in privileged modes.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.7.7 c2、c3、c5、および c6、PMSA のサポート

メモリ保護の一般的な原則は ARMv4 および ARMv5 でも同じですが、保護メモリの CP15 サポートは ARMv6 および ARMv7 のプログラミングモデルとは異なります。メモリ領域は、構成可能なベースアドレスとサイズ属性を持ちます。キャッシュ可能属性、バッファ可能属性、および領域間のアクセス許可について記述するレジスタも存在します。詳細については、*メモリモデルとメモリ順序* P. 付録 H-10 を参照して下さい。

ARMv4 および ARMv5 では、固定された数のメモリ領域がサポートされ、これらは次のいずれかです。

- 8 つの統一メモリ領域
- 8 つのデータ領域と 8 つの命令領域

PMSA レジスタのサポートを、表 H-19 に示します。

表 H-19 PMSA レジスタのサポート

レジスタ	CRn	opc1	CRm	opc2
データまたは統一キャッシュ可能レジスタ、DCR	c2	0	c0	0
命令キャッシュ可能レジスタ、ICR	c2	0	c0	1
データまたは統一バッファ可能レジスタ、DBR	c3	0	c0	0
データまたは統一アクセス許可レジスタ、DAPR	c5	0	c0	0
命令アクセス許可レジスタ、IAPR	c5	0	c0	1
データまたは統一拡張アクセス許可レジスタ、DEAPR	c5	0	c0	2
命令拡張アクセス許可レジスタ、IEAPR	c5	0	c0	3
データまたは統一メモリ領域レジスタ、DMRR0 ～ DMRR7	c6	0	c0 ～ c7[a]	0
命令メモリ領域レジスタ、IMRR0 ～ IMRR7	c6	0	c0 ～ c7[a]	1

a. `<CRm>` により、領域が選択されます。例えば、`<CRm>` == 6 の場合、領域 6 の領域レジスタ（DMRR6 または IMRR6）が選択されます。

実装に、命令アクセスとデータアクセスの両方に適用される単一の保護領域のセットが存在する場合は、`<opc2>` の偶数値を使用してアクセスされるレジスタのみが存在します。独立したデータ領域と命令領域がサポートされる場合、拡張アクセス許可レジスタを例外として、データに関連するレジスタでは `<opc2>` == 0 で、命令に関連するレジスタでは `<opc2>` == 1 です。PMSA レジスタはすべて 32 ビットレジスタで、特権モードでのみアクセス可能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c2, Memory Region Cacheability Registers (DCR and ICR)

The two Memory Region Cacheability Registers are:

- The Data or unified Cacheability Register, DCR.
- The Instruction Cacheability Register, ICR. The ICR is implemented only when the processor implements separate data and instruction memory protection region definitions.

A Memory Region Cacheability Registers holds a Cacheability bit, C, for each of the eight memory protection regions.

The format of a Memory Region Cacheability Register is:

31 … 8	7	6	5	4	3	2	1	0
Reserved	C7	C6	C5	C4	C3	C2	C1	C0

Bits [31:8] Reserved. UNK/SBZP.

Cn, bit [n], for n = 0 to 7

Cacheability bit, C, for memory protection region n.

Accessing the Memory Region Cacheability Registers

To access the Memory Region Cacheability Registers you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c2, `<CRm>` set to c0, and `<opc2>` set to:

- 0 for the DCR
- 1 for the IPR.

For example:

```
MRC p15,0,<Rt>,c2,c0,0    ; Read  CP15 Data or unified Region Cacheability Register
MCR p15,0,<Rt>,c2,c0,0    ; Write CP15 Data or unified Region Cacheability Register
MRC p15,0,<Rt>,c2,c0,1    ; Read  CP15 Instruction Region Cacheability Register
MCR p15,0,<Rt>,c2,c0,1    ; Write CP15 Instruction Region Cacheability Register
```

c3, Memory Region Bufferability Register (DBR)

The Memory Region Bufferability Register, DBR, holds Bufferability bit, B, for each of the eight data or unified memory protection regions.

Only data accesses are bufferable and therefore there is only a single Memory Region Bufferability Register, regardless of whether the implementation has a single set of protection regions, or separate protection region definitions for instruction and data accesses.

The format of the Memory Region Bufferability Register is:

31 … 8	7	6	5	4	3	2	1	0
Reserved	B7	B6	B5	B4	B3	B2	B1	B0

Bits [31:8] Reserved. UNK/SBZP.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

c2、メモリ領域キャッシュ可能レジスタ（DCR および ICR）

メモリ領域キャッシュ可能レジスタには、次の 2 つがあります。

- データまたは統一キャッシュ可能レジスタ、DCR
- 命令キャッシュ可能レジスタ、ICRICR は、プロセッサに独立したデータおよび命令のメモリ保護領域定義が実装されている場合にのみ実装されます。

メモリ領域キャッシュ可能レジスタには、8 つのメモリ保護領域のそれぞれについて、キャッシュ可能ビット (C) が保持されます。

メモリ領域キャッシュ可能レジスタのフォーマットは次のとおりです。

31 … 8	7	6	5	4	3	2	1	0
予約	C7	C6	C5	C4	C3	C2	C1	C0

ビット [31:8] 予約。UNK/SBZP。

Cn、ビット [n](n = 0 ～ 7)

メモリ保護領域 n のキャッシュ可能ビット (C)。

メモリ領域キャッシュ可能レジスタへのアクセス

メモリ領域キャッシュ可能レジスタにアクセスするには、`<opc1>` を 0、`<CRn>` を c2、`<CRm>` を c0、`<opc2>` を次の値に設定して、CP15 レジスタを読み出し / 書き込みします。

- DCR の場合、0
- IPR の場合、1

次に例を示します。

```
MRC p15,0,<Rt>,c2,c0,0    ; Read  CP15 Data or unified Region Cacheability Register
MCR p15,0,<Rt>,c2,c0,0    ; Write CP15 Data or unified Region Cacheability Register
MRC p15,0,<Rt>,c2,c0,1    ; Read  CP15 Instruction Region Cacheability Register
MCR p15,0,<Rt>,c2,c0,1    ; Write CP15 Instruction Region Cacheability Register
```

c3、メモリ領域バッファ可能レジスタ (DBR)

メモリ領域バッファ可能レジスタ (DBR) には、8 つのデータまたは統一メモリ保護領域のそれぞれについて、バッファ可能ビット (B) が保持されます。

データアクセスのみがバッファ可能なため、実装に存在する保護領域のセットが 1 つだけか、命令とデータへのアクセス用に独立した保護領域の定義が存在するかにかかわらず、メモリ領域バッファ可能レジスタは 1 つだけです。

メモリ領域バッファ可能レジスタのフォーマットは次のとおりです。

31 … 8	7	6	5	4	3	2	1	0
予約	B7	B6	B5	B4	B3	B2	B1	B0

ビット [31:8] 予約。UNK/SBZP。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Bn, bit [n], for n = 0 to 7

Bufferability bit, B, for memory protection region n.

Accessing the Memory Region Bufferability Register

To access the Memory Region Bufferability Register you read or write the CP15 registers with <opc1> set to 0, <CRn> set to c3, <CRm> set to c0, and <opc2> set to 0. For example:

```
MRC p15,0,<Rt>,c3,c0,0    ; Read  CP15 Data or unified Region Bufferability Register
MCR p15,0,<Rt>,c3,c0,0    ; Write CP15 Data or unified Region Bufferability Register
```

c5, Memory Region Access Permissions Registers (DAPR and IAPR)

The two Memory Region Access Permissions Registers are:

- The Data or unified Access Permissions Register, DAPR.
- The Instruction Access Permissions Register, IAPR. The IAPR is implemented only when the processor implements separate data and instruction memory protection region definitions.

A Memory Region Access Permissions Register hold the access permission bits AP[1:0] for each of the eight memory protection regions.

The format of a Memory Region Access Permissions Register is:

31 … 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	AP7		AP6		AP5		AP4		AP3		AP2		AP1		AP0	

Bits [31:16] Reserved. UNK/SBZP.

APn, bits [2n+1:2n], for n = 0 to 7

Access permission bits AP[1:0] for memory protection region n.

For details of the significance and encoding of these bits see Table H-10 on page AppxH-29.

If the implementation does not permit the requested type of access, it signals an abort to the processor.

Accessing the Memory Region Access Permissions Registers

To access the Memory Region Access Permissions Registers you read or write the CP15 registers with <opc1> set to 0, <CRn> set to c5, <CRm> set to c0, and <opc2> set as follows:

- 0 if there is only a single set of protection regions
- when there are separate memory protection regions for data and instructions:
  - — 0 to access the Data Region Access Permissions Register
  - — 1 to access the Instruction Region Access Permissions Register.

For example:

```
MRC p15,0,<Rt>,c5,c0,0    ; Read  CP15 Data or unified Region Access Permissions Register
MCR p15,0,<Rt>,c5,c0,0    ; Write CP15 Data or unified Region Access Permissions Register
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Bn、ビット [n](n = 0 ～ 7)

メモリ保護領域 n のバッファ可能ビット (B)。

メモリ領域バッファ可能レジスタへのアクセス

メモリ領域バッファ可能レジスタにアクセスするには、<opc1> を 0、<CRn> を c3、<CRm> を c0、<opc2> を 0 に設定して、CP15 レジスタを読み出し / 書き込みします。次に例を示します。

```
MRC p15,0,<Rt>,c3,c0,0    ; Read  CP15 Data or unified Region Bufferability Register
MCR p15,0,<Rt>,c3,c0,0    ; Write CP15 Data or unified Region Bufferability Register
```

c5、メモリ領域アクセス許可レジスタ（DAPR および IAPR）

メモリ領域アクセス許可レジスタには、次の 2 つがあります。

- データまたは統一アクセス許可レジスタ (DAPR)
- 命令アクセス許可レジスタ (IAPR)。IAPR は、プロセッサに独立したデータおよび命令のメモリ保護領域定義が実装されている場合にのみ実装されます。

メモリ領域アクセス許可レジスタには、8 つのメモリ保護領域のそれぞれについてアクセス許可ビット (AP[1:0]) が保持されます。

メモリ領域アクセス許可レジスタのフォーマットは次のとおりです。

31 … 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
予約	AP7		AP6		AP5		AP4		AP3		AP2		AP1		AP0	

ビット [31:16] 予約。UNK/SBZP。

APn、ビット [2n + 1:2n](n = 0 ～ 7)

メモリ保護領域 n のアクセス許可ビット (AP[1:0])。

これらのビットの上位下位およびエンコードの詳細については、P. 付録 H-29 の表 H-10 を参照して下さい。

要求されたタイプのアクセスが実装で許可されていない場合は、プロセッサにアボートが通知されます。

メモリ領域アクセス許可レジスタへのアクセス

メモリ領域アクセス許可レジスタにアクセスするには、<opc1> を 0、<CRn> を c5、<CRm> を c0、<opc2> を次の値に設定して、CP15 レジスタを読み出し / 書き込みします。

- 単一の保護領域のセットのみが存在する場合、0
- データおよび命令について独立のメモリ保護領域が存在する場合、次のようになります。
  - – データ領域アクセス許可レジスタにアクセスする場合、0
  - – 命令領域アクセス許可レジスタにアクセスする場合、1

次に例を示します。

```
MRC p15,0,<Rt>,c5,c0,0    ; Read  CP15 Data or unified Region Access Permissions Register
MCR p15,0,<Rt>,c5,c0,0    ; Write CP15 Data or unified Region Access Permissions Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

```
MRC p15,0,<Rt>,c5,c0,1    ; Read  CP15 Instruction Region Access Permissions Register
MCR p15,0,<Rt>,c5,c0,1    ; Write CP15 Instruction Region Access Permissions Register
```

c5, Memory Region Extended Access Permissions Registers (DEAPR and IEAPR)

The two Memory Region Extended Access Permissions Registers are:

- The Data or unified Extended Access Permissions Register, DEAPR.
- The Instruction Extended Access Permissions Register, IEAPR. The IEAPR is implemented only when the processor implements separate data and instruction memory protection region definitions.

Whether an implementation includes Extended Access Permissions Registers is IMPLEMENTATION DEFINED.

A Memory Region Extended Access Permissions Registers hold the access permission bits AP[3:0] for each of the eight memory protection regions.

The format of a Memory Region Access Permissions Register is:

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
AP7	AP6	AP5	AP4	AP3	AP2	AP1	AP0

APn, bits [4n+3:4n], for n = 0 to 7

Access permission bits AP[3:0] for memory protection region n.

For details of the significance of these bits see Table H-11 on page AppxH-29.

If the implementation does not permit the requested type of access, it signals an abort to the processor.

Accessing the Memory Region Extended Access Permissions Registers

To access the Memory Region Extended Access Permissions Registers you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c5, `<CRm>` set to c0, and `<opc2>` set as follows:

- 2 if there is only a single set of protection regions
- when there are separate memory protection regions for data and instructions:
  - — 2 to access the Data Region Extended Access Permissions Register
  - — 3 to access the Instruction Region Extended Access Permissions Register.

For example:

```
MRC p15,0,<Rt>,c5,c0,2   ; Read  CP15 Data or unified Region Extended Access Permissions Register
MCR p15,0,<Rt>,c5,c0,2   ; Write CP15 Data or unified Region Extended Access Permissions Register
MRC p15,0,<Rt>,c5,c0,3   ; Read  CP15 Instruction Region Extended Access Permissions Register
MCR p15,0,<Rt>,c5,c0,3   ; Write CP15 Instruction Region Extended Access Permissions Register
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

```
MRC p15,0,<Rt>,c5,c0,1    ; Read  CP15 Instruction Region Access Permissions Register
MCR p15,0,<Rt>,c5,c0,1    ; Write CP15 Instruction Region Access Permissions Register
```

c5、メモリ領域拡張アクセス許可レジスタ（DEAPR および IEAPR）

メモリ領域拡張アクセス許可レジスタには、次の 2 つがあります。

- データまたは統一拡張アクセス許可レジスタ、DEAPR
- 命令拡張アクセス許可レジスタ、IEAPR。IEAPR は、プロセッサに独立したデータおよび命令のメモリ保護領域定義が実装されている場合にのみ実装されます。

実装に拡張アクセス許可レジスタが含まれるかどうかは実装定義です。

メモリ領域拡張アクセス許可レジスタには、8 つのメモリ保護領域のそれぞれについてアクセス許可ビット (AP[3:0]) が保持されます。

メモリ領域拡張アクセス許可レジスタのフォーマットは次のとおりです。

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
AP7	AP6	AP5	AP4	AP3	AP2	AP1	AP0

APn、ビット [4n + 3:4n](n = 0 ～ 7)

メモリ保護領域 n のアクセス許可ビット (AP[3:0])。

これらのビットの上位下位の詳細については、P. 付録 H-29 の表 H-11 を参照して下さい。

要求されたタイプのアクセスが実装で許可されていない場合は、プロセッサにアボートが通知されます。

メモリ領域拡張アクセス許可レジスタへのアクセス

メモリ領域拡張アクセス許可レジスタにアクセスするには、`<opc1>` を 0、`<CRn>` を c5、`<CRm>` を c0、`<opc2>` を次の値に設定して、CP15 レジスタを読み出し / 書き込みします。

- 単一の保護領域のセットのみが存在する場合、2
- データおよび命令について独立のメモリ保護領域が存在する場合、次のようになります。
  - ー データ領域拡張アクセス許可レジスタにアクセスする場合、2
  - ー 命令領域拡張アクセス許可レジスタにアクセスする場合、3

次に例を示します。

```
MRC p15,0,<Rt>,c5,c0,2   ; Read  CP15 Data or unified Region Extended Access Permissions Register
MCR p15,0,<Rt>,c5,c0,2   ; Write CP15 Data or unified Region Extended Access Permissions Register
MRC p15,0,<Rt>,c5,c0,3   ; Read  CP15 Instruction Region Extended Access Permissions Register
MCR p15,0,<Rt>,c5,c0,3   ; Write CP15 Instruction Region Extended Access Permissions Register
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c6, Memory Region registers (DMRR0-DMRR7 and IMRR0-IMRR7)

The Memory Region registers define the MPU memory regions as follows:

- If an implementation supports only a single set of memory region definitions that apply to both data and instruction accesses, it must provide a single set of eight Data or unified Memory Region Registers, DMRR0-DMRR7.
- If an implementation supports separate memory region definitions for data and instruction accesses, it must provide two sets of eight Memory Region Registers:
  - — eight Data or unified Memory Region Registers, DMRR0-DMRR7
  - — eight Instruction Memory Region Registers, IMRR0-IMRR7.

Each Memory Region register:

- defines a single memory region by specifying its base address and size
- includes an enable bit for the associated memory region.

The format of a Memory Region register is:

31 … 12	11 … 6	5 … 1	0
Region base address	Reserved	Size	En

Region base address, bits [31:12]

Bits [31:12] of the base address for the region. Bits [11:0] of the address must be zero. Therefore, the smallest region that can be defined is 4KB. Regions must be aligned appropriately, and so for regions larger than 4KB the least significant bits of this field must be zero. For more information, see the description of the Size field.

Bits [11:6] Reserved. UNK/SBZP.

Size, bits [5:1]

Encodes the size of the region. Table H-20 shows the permitted encodings for this field.

Table H-20 MPU Region size encoding

Encoding	Region size	Base address constraints
0b01011	4KB	None
0b01100	8KB	Register bit [12] must be zero
0b01101	16KB	Register bits [13:12] must be zero
0b01110	32KB	Register bits [14:12] must be zero
0b01111	64KB	Register bits [15:12] must be zero
0b10000	128KB	Register bits [16:12] must be zero
0b10001	256KB	Register bits [17:12] must be zero

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

c6、メモリ領域レジスタ（DMRR0 ～ DMRR7 および IMRR0 ～ IMRR7）

メモリ領域レジスタは、MPU メモリ領域を次のように定義します。

- 実装で、データと命令のアクセスの両方に適用される単一のメモリ領域定義のセットのみがサポートされる場合、8 つのデータまたは統一メモリ領域レジスタ (DMRR0 ～ DMRR7) の単一のセットを提供する必要があります。
- 実装で、データと命令のアクセスについて個別のメモリ領域定義がサポートされる場合、8 つのメモリ領域レジスタのセットを 2 つ提供する必要があります。
  - – 8 つのデータまたは統一メモリ領域レジスタ、DMRR0 ～ DMRR7
  - – 8 つの命令メモリ領域レジスタ、IMRR0 ～ IMRR7

それぞれのメモリ領域レジスタには、次の特徴があります。

- ベースアドレスとサイズを指定して、単一のメモリ領域を定義します。
- 関連するメモリ領域のイネーブルビットが含まれています。

メモリ領域レジスタのフォーマットは次のとおりです。

31 … 12	11 … 6	5 … 1	0
領域のベースアドレス	予約	サイズ	En

領域のベースアドレス、ビット [31:12]

領域のベースアドレスのビット [31:12]。アドレスのビット [11:0] は 0 にする必要があります。このため、定義できる最小の領域は 4KB です。領域は適切にアラインする必要があるため、4KB より大きい領域の場合、このフィールドの最下位ビットを 0 にする必要があります。詳細については、サイズフィールドについての説明を参照して下さい。

ビット [11:6] . 予約。UNK/SBZP。

サイズ、ビット [5:1]

領域のサイズをエンコードします。このフィールドで許可されるエンコードを、表 H-20 に示します。

表 H-20 MPU 領域サイズのエンコード

エンコード	領域サイズ	ベースアドレスの制限
0b01011	4KB	なし
0b01100	8KB	レジスタのビット [12] は常に 0
0b01101	16KB	レジスタのビット [13:12] は常に 0
0b01110	32KB	レジスタのビット [14:12] は常に 0
0b01111	64KB	レジスタのビット [15:12] は常に 0
0b10000	128KB	レジスタのビット [16:12] は常に 0
0b10001	256KB	レジスタのビット [17:12] は常に 0

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table H-20 MPU Region size encoding (continued)

Encoding	Region size	Base address constraints
0b10010	512KB	Register bits [18:12] must be zero
0b10011	1MB	Register bits [19:12] must be zero
0b10100	2MB	Register bit [20:12] must be zero
0b10101	4MB	Register bits [21:12] must be zero
0b10110	8MB	Register bits [22:12] must be zero
0b10111	16MB	Register bits [23:12] must be zero
0b11000	32MB	Register bits [24:12] must be zero
0b11001	64MB	Register bits [25:12] must be zero
0b11010	128MB	Register bits [26:12] must be zero
0b11011	256MB	Register bits [27:12] must be zero
0b11100	512MB	Register bits [28:12] must be zero
0b11101	1G	Register bits [29:12] must be zero
0b11110	2GB	Register bits [30:12] must be zero
0b11111	4GB	Register bits [31:12] must be zero

Encodings not shown in the table are reserved. The effect of using a reserved value in this field is UNPREDICTABLE.

En, bit [0] Enable bit for the region:

0 Region is disabled

1 Region is enabled.

This field resets to zero. Therefore all MPU regions are disabled on reset.

The base address constraints given in Table H-20 on page AppxH-47 ensure that the specified region is correctly aligned in memory, so that its alignment is a multiple of the region size. If a base address is entered that does not follow these alignment constraints, behavior is UNPREDICTABLE.

Accessing the Region Access Permissions registers

To access the Region Access Permissions registers you read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c5, and:

- `<CRm>` set to indicate the region number, from `<CRm>` == c0 for memory region 0, to `<CRm>` == c7 for memory region 7

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 H-20 MPU 領域サイズのエンコード （続き）

エンコード	領域サイズ	ベースアドレスの制限
0b10010	512KB	レジスタのビット [18:12] は常に 0
0b10011	1MB	レジスタのビット [19:12] は常に 0
0b10100	2MB	レジスタのビット [20:12] は常に 0
0b10101	4MB	レジスタのビット [21:12] は常に 0
0b10110	8MB	レジスタのビット [22:12] は常に 0
0b10111	16MB	レジスタのビット [23:12] は常に 0
0b11000	32MB	レジスタのビット [24:12] は常に 0
0b11001	64MB	レジスタのビット [25:12] は常に 0
0b11010	128MB	レジスタのビット [26:12] は常に 0
0b11011	256MB	レジスタのビット [27:12] は常に 0
0b11100	512MB	レジスタのビット [28:12] は常に 0
0b11101	1GB	レジスタのビット [29:12] は常に 0
0b11110	2GB	レジスタのビット [30:12] は常に 0
0b11111	4GB	レジスタのビット [31:12] は常に 0

この表に示されていないエンコードは予約されています。このフィールドに予約されている値を使用した場合の影響は予測不能です。

En、ビット [0] 領域のイネーブルビット。

0 領域は非稼動です。

1 領域は稼動しています。

このフィールドのリセット時の値は 0 です。このため、MPU 領域はリセット時にすべて非稼動になります。

P. 付録 H-47 の表 H-20 に示されているベースアドレスの制限により、指定した領域のアライメントが領域サイズの倍数になるように、メモリで正確にアラインされることが保証されます。入力されたベースアドレスがこれらのアライメントの制限に従っていない場合、動作は予測不能です。

領域アクセス許可レジスタへのアクセス

領域アクセス許可レジスタにアクセスするには、`<opc1>` を 0、`<CRn>` を c5、他のパラメータを次のように設定して、CP15 レジスタを読み出し / 書き込みします。

- `<CRm>` には、領域番号を設定します。`<CRm>` == c0（メモリ領域 0 の場合）～ `<CRm>` == c7（メモリ領域 7 の場合）を指定します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- <opc2> set to:
  - — 0 if there is only a single set of protection region definitions
  - — 0 to access the Data Region Access Permissions Register when the data and instruction memory regions are defined separately
  - — 1 to access the Instruction Region Access Permissions Register when the data and instruction memory regions are defined separately.

For example:

```
MRC p15,0,<Rt>,c6,c0,0    ; Read  CP15 Data or unified Region Register, Region 0
MCR p15,0,<Rt>,c6,c0,0    ; Write CP15 Data or unified Region Register, Region 0
MRC p15,0,<Rt>,c6,c1,0    ; Read  CP15 Data or unified Region Register, Region 1
MCR p15,0,<Rt>,c6,c1,0    ; Write CP15 Data or unified Region Register, Region 1
MRC p15,0,<Rt>,c6,c2,1    ; Read  CP15 Instruction Memory Region Register, Region 2
MCR p15,0,<Rt>,c6,c2,1    ; Write CP15 Instruction Memory Region Register, Region 2
```

H.7.8 c7, Cache operations

Table H-21 shows the cache operation provision in ARMv4 and ARMv5. All cache operations are performed as MCR instructions and only operate on a level 1 cache associated with a specific processor. The equivalent operations in ARMv7 operate on multiple levels of cache. See *CP15 c7, Cache and branch predictor maintenance functions* on page B3-126.

Table H-21 Cache operation support

Operation	CRn	opc1	CRm	opc2
Invalidate instruction cache[a]	c7	0	c5	0
Invalidate instruction cache line by MVA[a]	c7	0	c5	1
Invalidate instruction cache line by set/way	c7	0	c5	2
Flush entire branch predictor array[a]	c7	0	c5	6
Flush branch predictor array entry by MVA[a]	c7	0	c5	7
Invalidate data cache	c7	0	c6	0
Invalidate data cache line by MVA[a]	c7	0	c6	1
Invalidate data cache line by set/way[a]	c7	0	c6	2
Invalidate unified cache, or instruction cache and data cache	c7	0	c7	0
Invalidate unified cache line by MVA	c7	0	c7	1
Invalidate unified cache line by set/way	c7	0	c7	2
Clean data cache line by MVA[a]	c7	0	c10	1

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- <opc2> は、次のように設定します。
  - – 単一の保護領域定義のセットのみが存在する場合、0
  - – データおよび命令メモリ領域が個別に定義されていて、データ領域アクセス許可レジスタにアクセスする場合、0
  - – データおよび命令メモリ領域が個別に定義されていて、命令領域アクセス許可レジスタにアクセスする場合、1

次に例を示します。

```
MRC p15,0,<Rt>,c6,c0,0    ; Read  CP15 Data or unified Region Register, Region 0
MCR p15,0,<Rt>,c6,c0,0    ; Write CP15 Data or unified Region Register, Region 0
MRC p15,0,<Rt>,c6,c1,0    ; Read  CP15 Data or unified Region Register, Region 1
MCR p15,0,<Rt>,c6,c1,0    ; Write CP15 Data or unified Region Register, Region 1
MRC p15,0,<Rt>,c6,c2,1    ; Read  CP15 Instruction Memory Region Register, Region 2
MCR p15,0,<Rt>,c6,c2,1    ; Write CP15 Instruction Memory Region Register, Region 2
```

H.7.8 c7、キャッシュ操作

ARMv4 および ARMv5 でのキャッシュ操作機構を、表 H-21 に示します。すべてのキャッシュ操作は MCR 命令として実行され、特定のプロセッサに関連付けられているレベル 1 キャッシュでのみ動作します。等価な ARMv7 の操作は、キャッシュの複数のレベルで動作します。*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126 を参照して下さい。

表 H-21 キャッシュ操作のサポート

動作	CRn	opc1	CRm	opc2
命令キャッシュの無効化[a]	c7	0	c5	0
MVA による命令キャッシュラインの無効化[a]	c7	0	c5	1
セット / ウェイによる命令キャッシュラインの無効化	c7	0	c5	2
分岐予測器アレイ全体のフラッシュ[a]	c7	0	c5	6
MVA による分岐予測器アレイエントリのフラッシュ[a]	c7	0	c5	7
データキャッシュの無効化	c7	0	c6	0
MVA によるデータ キャッシュラインの無効化[a]	c7	0	c6	1
セット / ウェイによるデータ キャッシュラインの無効化[a]	c7	0	c6	2
統一キャッシュ、または命令キャッシュとデータキャッシュの無効化	c7	0	c7	0
MVA による統一キャッシュラインの無効化	c7	0	c7	1
セット / ウェイによる統一キャッシュラインの無効化	c7	0	c7	2
MVA によるデータ キャッシュラインのクリーニング[a]	c7	0	c10	1
セット / ウェイによるデータ キャッシュラインのクリーニング[a]	c7	0	c10	2

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table H-21 Cache operation support (continued)

Operation	CRn	opc1	CRm	opc2
Clean data cache line by set/way[a]	c7	0	c10	2
Clean entire unified cache	c7	0	c11	0
Clean unified cache line by MVA[a]	c7	0	c11	1
Clean unified cache line by set/way	c7	0	c11	2
Prefetch instruction cache line by MVA[b]	c7	0	c13	1
Clean and Invalidate data cache line by MVA[a]	c7	0	c14	1
Clean and Invalidate data cache line by set/way[a]	c7	0	c14	2
Clean and Invalidate unified cache line by MVA	c7	0	c15	1
Clean and Invalidate unified cache line by set/way	c7	0	c15	2
Test and Clean data cache	c7	0	c10	3
Test and Clean and Invalidate data cache	c7	0	c14	3

a. These are the only cache operations available in ARMv7. The corresponding ARMv7 operations are multi-level operations, and the data cache operations are defined as data or unified cache operations.
b. Used with TLB lockdown. See *TLB lockdown procedure, using the by entry model* on page AppxH-61.

Test and clean operations

This scheme provides an efficient way to clean, or clean and invalidate, a complete data cache by executing an MRC instruction with the condition code flags as the destination. A global cache dirty status bit is written to the Z flag. How many lines are tested in each iteration of the instruction is IMPLEMENTATION DEFINED.

To clean an entire data cache with this method the following code loop can be used:

```
tc_loop   MRC p15, 0, APSR_nzcv, c7, c10, 3       ; test and clean
          BNE tc_loop
```

To clean and invalidate an entire data cache with this method, the following code loop can be used:

```
tci_loop  MRC p15, 0, APSR_nzcv, c7, c14, 3       ; test, clean and invalidate
          BNE tci_loop
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 H-21 キャッシュ操作のサポート （続き）

動作	CRn	opc1	CRm	opc2
統一キャッシュ全体のクリーニング	c7	0	c11	0
MVA による統一キャッシュラインのクリーニング[a]	c7	0	c11	1
セット / ウェイによる統一キャッシュラインのクリーニング	c7	0	c11	2
MVA による命令キャッシュラインのプリフェッチ[b]	c7	0	c13	1
MVA によるデータ キャッシュラインのクリーニングと無効化[a]	c7	0	c14	1
セット / ウェイによるデータ キャッシュラインのクリーニングと無効化[a]	c7	0	c14	2
MVA による統一キャッシュラインのクリーニングと無効化	c7	0	c15	1
セット / ウェイによる統一キャッシュラインのクリーニングと無効化	c7	0	c15	2
データキャッシュのテストとクリーニング	c7	0	c10	3
データキャッシュのテスト、クリーニング、無効化	c7	0	c14	3

a. ARMv7 で使用可能なキャッシュ操作はこれらだけです。対応する ARMv7 操作は複数レベルの操作で、データキャッシュ操作はデータまたは統一キャッシュの操作として定義されます。
b. TLB ロックダウンとともに使用されます。*エントリモデルを使用した TLB ロックダウンの手順* P. 付録 H-61 を参照して下さい。

テスト操作とクリーニング操作

この方式では、条件コードフラグをデスティネーションとして MRC 命令を実行することにより、データキャッシュ全体のクリーニング、またはクリーニングと無効化を効率的に行うことができます。グローバルなキャッシュダーティステータス ビットは、Z フラグに書き込まれます。命令の繰り返しのそれぞれでテストされるラインの数は実装定義です。

この方式を使用してデータキャッシュ全体をクリーニングするには、次のコードループを使用できます。

```
tc_loop   MRC p15, 0, APSR_nzcv, c7, c10, 3       ; test and clean
          BNE tc_loop
```

この方式を使用してデータキャッシュ全体のクリーニングと無効化を行うには、次のコードループを使用できます。

```
tci_loop  MRC p15, 0, APSR_nzcv, c7, c14, 3       ; test, clean and invalidate
          BNE tci_loop
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Level 2 cache support

The recommended method for adding closely coupled level 2 cache support from ARMv5TE is to define equivalent operations to the level 1 support with `<opc1>` == 1 in the appropriate `MCR` instructions. The operations in Table H-21 on page AppxH-49 that are supported are IMPLEMENTATION DEFINED.

H.7.9 c7, Miscellaneous functions

The Wait For Interrupt operation is used in some implementations as part of a power management support scheme. The operation is deprecated in ARMv6 and not supported in ARMv7 (it behaves as a `NOP` instruction).

Barrier operations are used for system correctness to ensure visibility of memory accesses to other agents in a system. For ARMv4 and ARMv5 the requirement for and use of barrier operations is IMPLEMENTATION DEFINED. Barrier functionality is formally defined as part of the memory architecture enhancements introduced in ARMv6.

Table H-22 summarizes the `MCR` instruction encoding details.

Table H-22 Memory barrier register support

Operation	CRn	opc1	CRm	opc2
Wait For Interrupt (CP15WFI)	c7	0	c0	4
Instruction Synchronization Barrier (CP15ISB)[a]	c7	0	c5	4
Data Synchronization Barrier (CP15DSB)[b]	c7	0	c10	4
Data Memory Barrier (CP15DMB)	c7	0	c10	5

a. This operation was previously known as *Prefetch Flush* (PF or PFF).
b. This operation was previously known as *Data Write Barrier* or *Drain Write Buffer* (DWB).

H.7.10 c8, VMSA TLB support

Table H-23 illustrates TLB operation provision in ARMv4 and ARMv5. All TLB operations are performed as `MCR` instructions and are a subset of the operations available in ARMv7. See *CP15 c8, TLB maintenance operations* on page B3-138.

Table H-23 TLB operation support

Operation	CRn	opc1	CRm	opc2
Invalidate Instruction TLB	c8	0	c5	0
Invalidate Instruction TLB Entry (by MVA)	c8	0	c5	1
Invalidate Data TLB	c8	0	c6	0

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

レベル 2 キャッシュのサポート

ARMv5TE およびそれ以降で密結合されたレベル 2 キャッシュのサポートを追加する場合の推奨される方法は、該当する `MCR` 命令で `<opc1>` == 1 を指定し、レベル 1 のサポートと等価な操作を定義することです。P. 付録 H-49 の表 H-21 の操作のうち、実装でどれがサポートされるかは実装定義です。

H.7.9 c7、その他の機能

割り込み待ち操作は、一部の実装で電力管理サポート方式の一部として使用されます。この操作は ARMv6 では非推奨で、ARMv7 ではサポートされていません（`NOP` 命令として動作します）。

バリア操作は、システムの正確性の目的で、システム内の他のエージェントからメモリアクセスが可視であることを保証するために使用されます。ARMv4 および ARMv5 では、バリア操作の要件およびその使用は実装定義です。バリア機能は、ARMv6 で導入されたメモリアーキテクチャ拡張機能の一部として正式に定義されています。

`MCR` 命令のエンコードの詳細を、表 H-22 に要約します。

表 H-22 メモリバリア レジスタのサポート

動作	CRn	opc1	CRm	opc2
割り込み待ち(CP15WFI)	c7	0	c0	4
命令同期バリア(CP15ISB)[a]	c7	0	c5	4
データ同期バリア(CP15DSB)[b]	c7	0	c10	4
データメモリ バリア(CP15DMB)	c7	0	c10	5

a. この操作は、以前は*プリフェッチフラッシュ*（PF または PFF）と呼ばれていました。
b. この操作は、以前は*データライト バリア*または*ライトバッファ ドレイン*(DWB) と呼ばれていました。

H.7.10 c8、VMSA TLB のサポート

ARMv4 および ARMv5 での TLB 操作の機構を、表 H-23 に示します。すべての TLB 操作は `MCR` 命令として実行され、ARMv7 で使用可能な操作のサブセットです。*CP15 c8、TLB 保守操作* P. B3-138 を参照して下さい。

表 H-23 TLB 操作のサポート

動作	CRn	opc1	CRm	opc2
命令 TLB の無効化	c8	0	c5	0
MVA による命令 TLB エントリの無効化	c8	0	c5	1
データ TLB の無効化	c8	0	c6	0
MVA によるデータ TLB エントリの無効化	c8	0	c6	1
統一 TLB の無効化	c8	0	c7	0
MVA による統一 TLB エントリの無効化	c8	0	c7	1

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table H-23 TLB operation support (continued)

Operation	CRn	opc1	CRm	opc2
Invalidate Data TLB Entry (by MVA)	c8	0	c6	1
Invalidate Unified TLB	c8	0	c7	0
Invalidate Unified TLB Entry (by MVA)	c8	0	c7	1

H.7.11 c9, cache lockdown support

One problem with caches is that although they normally improve average access time to data and instructions, they usually increase the worst-case access time. This is because:

- There is a delay before the system determines that a cache miss has occurred and starts the main memory access.
- If a Write-Back cache is being used, there might be more delay because of the requirement to store the contents of the cache line that is being reallocated.
- A whole cache line is loaded from main memory, not only the data requested by the ARM processor.

In real-time applications, this increase in the worst-case access time can be significant.

Cache lockdown is an optional feature designed to alleviate this. It enables critical code and data, for example high priority interrupt routines and the data they access, to be loaded into the cache in such a way that the cache lines containing them are not subsequently reallocated. This ensures that all subsequent accesses to this code and data are cache hits and therefore complete as quickly as possible.

The ARM architecture specifies four formats for the cache lockdown mechanism. These are known as Format A, Format B, Format C, and Format D. The Cache Type Register contains information on the lockdown mechanism adopted. See *c0, Cache Type Register (CTR)* on page AppxH-35.

Formats A, B, and C all operate on cache ways. Format D is a cache entry locking mechanism. Table H-24 summarizes the CP15 provisions for format A, B, C, and D lockdown mechanisms.

From ARMv7, cache lockdown is IMPLEMENTATION DEFINED with no recommended formats or mechanisms on how it is achieved other than reserved CP15 register space. See *Cache lockdown* on page B2-8 and *CP15 c9, Cache and TCM lockdown registers and performance monitors* on page B3-141.

Table H-24 cache lockdown register support

Register or operation	Lockdown formats	CRn	opc1	CRm	opc2
Data or unified Cache Lockdown Register, DCLR	A, B, and C	c9	0	c0	0
Instruction Cache Lockdown Register, ICLR	A, B, and C	c9	0	c0	1
Fetch and lock instruction cache line	D	c9	0	c5	0

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

H.7.11 c9、キャッシュロックダウンのサポート

キャッシュに関する問題の 1 つは、キャッシュにより一般にデータおよび命令への平均アクセス時間は短縮するが、ワーストケースのアクセス時間は増大するということです。これには、次のような理由があります。

- キャッシュミスが発生したことをシステムが確認し、メインメモリへのアクセスを開始するまでに遅延が発生する。
- ライトバックキャッシュが使用されている場合、再割り当てされるキャッシュラインの内容を保管する必要があり、それによる余分な遅延が発生する可能性があるため。
- ARM プロセッサによって要求されたデータだけではなく、1 つのキャッシュラインの全体がメインメモリからロードされる。

リアルタイムアプリケーションでは、ワーストケースでのアクセス時間の増加が問題になる場合があります。

キャッシュロックダウンは、これを軽減するために設計されたオプションの機能です。この機能を使用すると、例えば優先度の高い割り込みルーチンや、そのルーチンがアクセスするデータなどの重要なコードやデータを、それらが含まれるキャッシュラインが以後に再割り当てされないようにキャッシュにロードできます。これにより、このコードやデータに対する以降のアクセスが必ずキャッシュヒットになるため、可能な限り高速に実行されることが保証されます。

ARM アーキテクチャでは、キャッシュロックダウン機構で 4 つのフォーマットが指定されています。これらは、フォーマット A、フォーマット B、フォーマット C、およびフォーマット D と呼ばれます。キャッシュタイプ レジスタには、採用されているロックダウン機構に関する情報が含まれています。*c0、キャッシュタイプ レジスタ (CTR)* P. 付録 H- 35 を参照して下さい。

フォーマット A、B、C は、すべてキャッシュウェイで動作します。フォーマット D は、キャッシュエントリのロック機構です。フォーマット A、B、C、D のロックダウン機構に対する CP15 機構の要約を、表 H- 24 に示します。

ARMv7 およびそれ以降では、キャッシュロックダウンは実装定義で、予約 CP15 レジスタ空間以外で実行される場合の推奨されるフォーマットや機構は存在しません。*キャッシュロックダウン* P. B2- 8 および *CP15 c9、キャッシュおよび TCM ロックダウンレジスタとパフォーマンスモニタ* P. B3- 141 を参照して下さい。

表 H- 24 キャッシュロックダウン レジスタのサポート

レジスタまたは操作	ロックダウンのフォーマット	CRn	opc1	CRm	opc2
データまたは統一キャッシュロックダウン レジスタ、DCLR	A、B、C	c9	0	c0	0
命令キャッシュロックダウン レジスタ、ICLR	A、B、C	c9	0	c0	1
命令キャッシュラインのフェッチとロック	D	c9	0	c5	0
命令キャッシュのロック解除	D	c9	0	c5	1
フォーマット D のデータまたは統一キャッシュのロックダウン レジスタ、DCLR2	D	c9	0	c6	0
データキャッシュのロック解除	D	c9	0	c6	1

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table H-24 cache lockdown register support (continued)

Register or operation	Lockdown formats	CRn	opc1	CRm	opc2
Unlock instruction cache	D	c9	0	c5	1
Format D Data or unified Cache Lockdown Register, DCLR2	D	c9	0	c6	0
Unlock data cache	D	c9	0	c6	1

General conditions applying to Format A, B, and C lockdown

The instructions used to access the CP15 c9 lockdown registers are as follows:

```
MCR p15, 0, <Rt>, c9, c0, 0     ; write Data or unified Cache Lockdown Register
MRC p15, 0, <Rt>, c9, c0, 0     ; read Data or unified Cache Lockdown Register
MCR p15, 0, <Rt>, c9, c0, 1     ; write Instruction Cache Lockdown Register
MRC p15, 0, <Rt>, c9, c0, 1     ; read Instruction Cache Lockdown Register
```

Formats A, B, and C all use cache ways for lockdown granularity. Granularity is defined by the *lockdown block*, and a cache locking scheme can use any number of lockdown blocks from 1 to (ASSOCIATIVITY-1).

If N lockdown blocks are locked down, they have indices 0 to N-1, and lockdown blocks N to (ASSOCIATIVITY-1) are available for normal cache operation.

A cache way based lockdown implementation must not lock down the entire cache. At least one cache way must be left for normal cache operation, otherwise behavior is UNPREDICTABLE.

The lockdown blocks are indexed from 0 to (ASSOCIATIVITY-1). The cache lines in a lockdown block are chosen to have the same WAY number as the lockdown block index value. So lockdown block n consists of the cache line with index n from each cache set, and n takes the values from n == 0 to n == (ASSOCIATIVITY-1).

Where `NSETS` is the number of sets, and `LINELEN` is the cache line length, each lockdown block can hold NSETS memory cache lines, provided each of the memory cache lines is associated with a different cache set. ARM recommends that systems are designed so that each lockdown block contains a set of NSETS consecutive memory cache lines. This is NSETS × LINELEN consecutive memory locations, starting at a cache line boundary. Such sets are easily identified and are guaranteed to consist of one cache line associated with each cache set.

Formats A and B lockdown

Formats A and B use a WAY field that is chosen to be wide enough to hold the way number of any lockdown block. Its width, W, is given by $W = \log_2(\text{ASSOCIATIVITY})$, rounded up to the nearest integer if necessary.

The format of a Format A lockdown register is:

31 – 32–W	31–W – 0
WAY	UNK/SBZ

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

フォーマット A、B、C のロックダウンに適用される一般的な条件

CP15 c9 ロックダウンレジスタにアクセスするため使用される命令は、次のとおりです。

```
MCR p15, 0, <Rt>, c9, c0, 0     ; write Data or unified Cache Lockdown Register
MRC p15, 0, <Rt>, c9, c0, 0     ; read Data or unified Cache Lockdown Register
MCR p15, 0, <Rt>, c9, c0, 1     ; write Instruction Cache Lockdown Register
MRC p15, 0, <Rt>, c9, c0, 1     ; read Instruction Cache Lockdown Register
```

フォーマット A、B、C のすべてについて、ロックダウンの粒度としてキャッシュウェイが使用されます。粒度は*ロックダウンブロック*により定義され、キャッシュロック方式では 1 ～ (ASSOCIATIVITY − 1) の任意の数のロックダウンブロックを使用できます。

N 個のロックダウンブロックがロックダウンされる場合、インデクスは 0 ～ N − 1 で、通常のキャッシュ操作では N ～ (ASSOCIATIVITY − 1) のロックダウンブロックが使用可能です。

ロックダウン実装に基づいたキャッシュウェイでは、キャッシュ全体をロックダウンすることはできません。通常のキャッシュ操作用に最低 1 つのキャッシュウェイを残しておく必要があります。残されていない場合、動作は予測不能です。

ロックダウンブロックには、0 ～ (ASSOCIATIVITY − 1) のインデクスが付けられます。ロックダウンブロック内のキャッシュラインは、ロックダウンブロックのインデクス値と同じ WAY 番号を持つように選択されます。このため、ロックダウンブロック n は、各キャッシュセットでインデクス n が付けられたキャッシュラインから構成されます。ここで n の値は、n == 0 ～ n == (ASSOCIATIVITY − 1) です。

`NSETS` はセットの数で、`LINELEN` はキャッシュライン長です。メモリ キャッシュラインのそれぞれが別のキャッシュセットに関連付けられている場合、各ロックダウンブロックは NSETS 個のメモリ キャッシュラインを保持できます。各ロックダウンブロックに、NSETS 個の連続したメモリ キャッシュラインが保持されるようにシステムを設計することをお勧めします。これは NSETS × LINELEN 個の連続したメモリ位置で、キャッシュラインの境界から開始されます。このセットは簡単に識別され、各キャッシュセットに関連付けられた 1 つのキャッシュラインから構成されることが保証されます。

フォーマット A および B のロックダウン

フォーマット A および B では、いずれのロックダウンブロックのウェイ数も保持することができるように十分な幅に選択されている WAY フィールドが使用されます。その幅 W は、$W = \log_2(\text{ASSOCIATIVITY})$ で与えられ、必要な場合はより大きな最も近い整数に丸められます。

フォーマット A のロックダウンレジスタのフォーマットは次のとおりです。

31 – 32 − W	31 − W – 0
WAY	UNK/SBZ

フォーマット A のレジスタを読み出すと、最後に書き込まれた値が返されます。

フォーマット A のレジスタに書き込むと、次の動作が行われます。

- 各キャッシュセットの次のキャッシュミスの際に、キャッシュラインがそのキャッシュセットに指定された WAY で置き換えられます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Reading a Format A register returns the value last written to it.

Writing a Format A register has the following effects:

- The next cache miss in each cache set replaces the cache line with the specified WAY in that cache set.
- The replacement strategy for the cache is constrained so that it can only select cache lines with the specified WAY and higher until the register is written again.

The format of a Format B lockdown register is:

31	30 … W	W–1 … 0
L	UNK/SBZ	WAY

Reading a Format B register returns the value last written to it.

Writing a Format B register has the following effects:

- If L == 1, all cache misses replace the cache line with the specified WAY in the relevant cache set until the register is written again.
- If L == 0:
  - — If the previous value of L was 0, and the previous value of WAY is smaller than the new value, the behavior is UNPREDICTABLE.
  - — If the previous value of L was not 0, the replacement strategy for the cache is constrained so that it can only select cache lines with the specified WAY and higher until the register is written again.

Format A and B cache lockdown procedure

The procedure for locking down N lockdown blocks is as follows:

1. Ensure that no processor exceptions can occur during the execution of this procedure, for example by disabling interrupts. If for some reason this is not possible, all code and data used by any exception handlers that can get called must be treated as code and data used by this procedure for the purpose of steps 2 and 3.
2. If an instruction cache or a unified cache is being locked down, ensure that all the code executed by this procedure is in an Non-cacheable area of memory.
3. If a data cache or a unified cache is being locked down, ensure that all data used by the following code is in an Non-cacheable area of memory, apart from the data that is to be locked down.
4. Ensure that the data or instructions that are to be locked down are in a Cacheable area of memory.
5. Ensure that the data or instructions that are to be locked down are not already in the cache, using cache clean, invalidate, or clean and invalidate instructions as appropriate.
6. For each value of i from 0 to N-1:
   a. Write to the CP15 c9 register with:
      - WAY == i, for Formats A and B
      - L == 1, for Format B only.
   b. For each of the cache lines to be locked down in lockdown block i:

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

- 再度レジスタに書き込みが行われるまで、指定した WAY 以上の値を持つキャッシュラインのみが選択されるように、キャッシュの置換方式が制限されます。

フォーマット B のロックダウンレジスタのフォーマットは次のとおりです。

31	30 … W	W - 1 … 0
L	UNK/SBZ	WAY

フォーマット B のレジスタを読み出すと、最後に書き込まれた値が返されます。

フォーマット B のレジスタに書き込むと、次の動作が行われます。

- L == 1 の場合、再度レジスタに書き込みが行われるまでに発生するすべてのキャッシュミスで、関連するキャッシュセット内でキャッシュラインが指定した WAY で置き換えられます。
- L == 0 の場合は次のようになります。
  - L の前の値が 0 で、WAY の前の値が新規の値よりも小さい場合、動作は予測不能です。
  - L の前の値が 0 ではない場合は、再度レジスタに書き込みが行われるまで、指定した WAY 以上の値を持つキャッシュラインのみが選択されるように、キャッシュの置換方式が制限されます。

フォーマット A および B のキャッシュロックダウンの手順

N 個のロックダウンブロックをロックダウンする手順は、次のとおりです。

1. 割り込みを不可能にするなどの方法で、この手順の実行中にプロセッサの例外が発生しないことを保証します。何らかの理由でこれが不可能な場合、呼び出される可能性がある例外ハンドラによって使用されるすべてのコードとデータは、ステップ 2 および 3 の目的のためにこの手順で使用されるコードおよびデータとして処理する必要があります。
2. 命令キャッシュまたは統一キャッシュをロックダウンする場合は、この手順で実行されるすべてのコードがメモリのキャッシュ不可領域にあることを確認します。
3. データキャッシュまたは統一キャッシュをロックダウンする場合は、ロックダウンするデータを除いて、以下のコードで使用されるすべてのデータがメモリのキャッシュ不可領域にあることを確認します。
4. ロックダウンされるデータまたは命令が、メモリのキャッシュ可能領域にあることを確認します。
5. 必要な場合はキャッシュのクリーニング、無効化、またはクリーニングと無効化の命令を使用して、ロックダウンされるデータまたは命令がまだキャッシュ内にないことを確認します。
6. 0 ～ N－1 の各 i 値について、次の処理を行います。
   a. 次の設定で CP15 c9 レジスタへの書き込みを行います。
      - WAY == i（フォーマット A および B の場合）
      - L == 1（フォーマット B の場合のみ）
   b. ロックダウンブロック i でロックダウンされるキャッシュラインのそれぞれについて、次の処理を行います。
      データキャッシュまたは統一キャッシュをロックダウンする場合は、LDR 命令を使用してメモリ キャッシュラインから 1 ワードをロードします。この操作により、メモリ キャッシュラインがキャッシュにロードされることが保証されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If a data cache or a unified cache is being locked down, use an LDR instruction to load a word from the memory cache line. This ensures that the memory cache line is loaded into the cache.

If an instruction cache is being locked down, use the CP15 c7 prefetch instruction cache line operation to fetch the memory cache line into the cache.

7. Write to the CP15 c9 register with:
   - WAY == N, for Formats A and B
   - L == 0, for Format B only.

Note

If the FCSE described in Appendix E *Fast Context Switch Extension (FCSE)* is being used, care must be taken in step 6b because:

- If a data cache or a unified cache is being locked down, the address used for the LDR instruction is subject to modification by the FCSE.
- If an instruction cache is being locked down, the address used for the CP15 c7 operation is treated as data and so is not subject to modification by the FCSE.

To minimize the possible confusion caused by this, ARM recommends that the lockdown procedure:

- starts by disabling the FCSE (by setting the PID to zero)
- where appropriate, generates modified virtual addresses itself by ORing the appropriate PID value into the top seven bits of the virtual addresses it uses.

Format A and B cache unlock procedure

To unlock the locked down portion of the cache, write to the CP15 c9 register with:

- WAY == 0, for Formats A and B
- L == 0, for Format B only.

Format C lockdown

Cache lockdown Format C is a different form of cache way based locking. It enables the allocation to each cache way to be disabled or enabled. This provides some additional control over the cache pollution caused by particular applications, in addition to a traditional lockdown function for locking critical regions into the cache.

A locking bit for each cache way determines whether the normal cache allocation mechanisms can access that cache way.

For caches of higher associativity, only cache ways 0 to 31 can be locked.

A maximum of N-1 ways of an N-way cache can be locked. This ensures that a normal cache line replacement can be performed. Handling a cache miss is UNPREDICTABLE if there are no cache ways that have L==0.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

命令キャッシュをロックダウンする場合は、CP15 c7 プリフェッチ命令キャッシュライン操作を使用して、メモリ キャッシュラインをキャッシュにフェッチします。

7. 次の設定で CP15 c9 レジスタへの書き込みを行います。
   - WAY == N（フォーマット A および B の場合）
   - L == 0（フォーマット B の場合のみ）

注

付録 E*「高速コンテキストスイッチ拡張機能 (FCSE)」*で説明されている FCSE が使用されている場合は、次のような理由から、ステップ 6b で注意が必要です。

- データキャッシュまたは統一キャッシュをロックダウンする場合、LDR 命令で使用されるアドレスは FCSE による変更の対象となる。
- 命令キャッシュをロックダウンする場合、CP15 c7 操作で使用されるアドレスはデータとして処理され、FCSE による変更の対象とならない。

このことにより発生する可能性のある混乱を最小限にするために、ロックダウン手順で次の処理を行うことをお勧めします。

- FCSE を非稼動（PID をゼロに設定）して手順を開始する。
- 必要な場合は、該当する PID 値と使用される仮想アドレスの上位 7 ビットとで OR 演算を行い、修飾仮想アドレスを生成する。

フォーマット A および B のキャッシュロック解除の手順

キャッシュのロックダウンされた部分をロック解除するには、次の設定で CP15 c9 レジスタへの書き込みを行います。

- WAY == 0（フォーマット A および B の場合）
- L == 0（フォーマット B の場合のみ）

フォーマット C のロックダウン

キャッシュロックダウンのフォーマット C は、キャッシュウェイ ベースのロックとは異なる形式になります。このフォーマットでは、各キャッシュウェイへの割り当てを可能または不可能に設定できます。これにより、重要な領域をキャッシュ内にロックする従来のロックダウン機能に加え、特定のアプリケーションで発生するキャッシュの汚染をある程度制御することができます。

各キャッシュウェイのロックビットにより、通常のキャッシュ割り当て機構でそのキャッシュウェイにアクセス可能かどうかが決定されます。

アソシエティビティの数がより多いキャッシュの場合は、キャッシュウェイ 0 ～ 31 のみをロックできます。

N ウェイキャッシュのうち最大 N－1 ウェイをロックできます。この操作により、通常のキャッシュライン置換が実行可能であることが保証されます。L == 0 のキャッシュウェイが存在しない場合、キャッシュミスの処理は予測不能です。

ロックダウンレジスタの全 32 ビットにより、関連するキャッシュウェイの L ビットが決定されます。<opc2> の値により、命令ロックダウンレジスタとデータロックダウン レジスタのどちらにアクセスするかが決定されます。

キャッシュロックダウン レジスタは通常、読み出し－変更－書き込みのシーケンスで変更されます。例えば、次のシーケンスでは、命令キャッシュのウェイ 0 について、L ビットが 1 にセットされます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The 32 bits of the lockdown register determine the L bit for the associated cache way. The value of <opc2> determines whether the instruction lockdown register or data lockdown register is accessed.

The cache lockdown register is normally modified in a read, modify, write sequence. For example, the following sequence sets the L bit to 1 for way 0 of the instruction cache:

```
; In the following code, <Rn> can be any register whose value does not need to be kept.
        MRC     p15, 0, <Rn>, c9, c0, 1
        ORR     <Rn>, <Rn>, #0x01
        MCR     p15, 0, <Rn>, c9, c0, 1         ; Set way 0 L bit for the instruction cache
```

The format of the Format C lockdown register is:

31	0
One L bit for each cache way	

Bits [31:0] The L bits for each cache way. If a cache way is not implemented, the L bit for that way is RAO/WI. Each bit relates to its corresponding cache way, that is bit N refers to way N.

- **0** Allocation to the cache way is determined by the standard replacement algorithm (reset state)
- **1** No Allocation is performed to this cache way.

The Format C lockdown register must only be changed when it is certain that all outstanding accesses that can cause a cache linefill have completed. For this reason, a Data Synchronization Barrier instruction must be executed before the lockdown register is changed.

Format C cache lock procedure

The procedure for locking down into a cache way i with N cache ways using Format C involves making it impossible to allocate to any cache way other than the target cache way i. The architecture defines the following method for locking data into the caches:

1. Ensure that no processor exceptions can occur during the execution of this procedure, for example by disabling interrupts. If for some reason this is not possible, all code and data used by any exception handlers that can get called must be treated as code and data used by this procedure for the purpose of steps 2 and 3.
2. If an instruction cache or a unified cache is being locked down, ensure that all the code executed by this procedure is in an Non-cacheable area of memory, including the Tightly Coupled Memory, or in an already locked cache way.
3. If a data cache or a unified cache is being locked down, ensure that all data used by the following code (apart from the data that is to be locked down) is in an Non-cacheable area of memory, including the Tightly Coupled Memory, or is in an already locked cache way.
4. Ensure that the data or instructions that are to be locked down are in a Cacheable area of memory.
5. Ensure that the data or instructions that are to be locked down are not already in the cache, using cache clean, invalidate, or clean and invalidate instructions as appropriate.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

```
; In the following code, <Rn> can be any register whose value does not need to be kept.
        MRC     p15, 0, <Rn>, c9, c0, 1
        ORR     <Rn>, <Rn>, #0x01
        MCR     p15, 0, <Rn>, c9, c0, 1         ; Set way 0 L bit for the instruction cache
```

フォーマット C のロックダウンレジスタのフォーマットは次のとおりです。

31	0
各キャッシュウェイについて 1 つの L ビット	

ビット [31:0] 各キャッシュウェイの L ビット。キャッシュウェイが実装されていない場合、そのウェイの L ビットは RAO/WI です。各ビットは対応するキャッシュウェイに関連付けられています。つまり、ビット N はウェイ N を指します。

- **0** キャッシュウェイへの割り当ては、標準の置換アルゴリズムにより決定されます (リセット時の状態)。
- **1** このキャッシュウェイに対する割り当ては実行されません。

フォーマット C のロックダウンレジスタは、キャッシュラインフィルを発生させる可能性がある未解決のアクセスがすべて確実に完了した後にのみ変更できます。このため、ロックダウンレジスタが変更される前にデータ同期バリア命令を実行する必要があります。

フォーマット C のキャッシュロックの手順

フォーマット C を使用して N 個のキャッシュウェイのうちキャッシュウェイ i をロックダウンする手順には、ターゲットのキャッシュウェイ i 以外のキャッシュウェイに割り当てることを不可能にする手順が含まれています。アーキテクチャでは、データをキャッシュにロックする次の方法が定義されています。

1. 割り込みを不可能にするなどの方法で、この手順の実行中にプロセッサの例外が発生しないことを保証します。何らかの理由でこれが不可能な場合、呼び出される可能性がある例外ハンドラによって使用されるすべてのコードとデータは、ステップ 2 および 3 の目的のためにこの手順で使用されるコードおよびデータとして処理する必要があります。
2. 命令キャッシュまたは統一キャッシュをロックダウンする場合は、この手順で実行されるすべてのコードが密結合メモリなどメモリのキャッシュ不可領域にあるか、またはすでにロックされているキャッシュウェイにあることを確認します。
3. データキャッシュまたは統一キャッシュをロックダウンする場合は、以下のコードで使用されるすべてのデータ（ロックダウンするデータを除く）が密結合メモリなどメモリのキャッシュ不可領域にあるか、またはすでにロックされているキャッシュウェイにあることを確認します。
4. ロックダウンされるデータまたは命令が、メモリのキャッシュ可能領域にあることを確認します。
5. 必要な場合はキャッシュのクリーニング、無効化、またはクリーニングと無効化の命令を使用して、ロックダウンされるデータまたは命令がまだキャッシュ内にないことを確認します。
6. ビット i を L = 0、他のすべてのビットを L = 1 に設定し、CRm == 0 を指定して CP15 c9 レジスタへの書き込みを行います。この操作により、ターゲットのキャッシュウェイ i に対する割り当てが許可されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

6. Write to the CP15 c9 register with CRm == 0, setting L=0 for bit i and L=1 for all other bits. This enables allocation to the target cache way i.

7. For each of the cache lines to be locked down in cache way i:
   - If a data cache or a unified cache is being locked down, use an LDR instruction to load a word from the memory cache line. This ensures that the memory cache line is loaded into the cache.
   - If an instruction cache is being locked down, use the CP15 c7 prefetch instruction cache line operation to fetch the memory cache line into the cache.

8. Write to the CP15 c9 register with CRm == 0, setting L = 1 for bit i and restoring all the other bits to the values they had before this routine was started.

Format C cache unlock procedure

To unlock the locked down portion of the cache, write to the CP15 c9 register, setting L == 0 for each bit.

Format D lockdown

This format locks individual L1 cache line entries rather than using a cache way scheme. The methods differ for the instruction and data caches.

The instructions used to access the CP15 c9 Format D Cache Lockdown Registers and operations are as follows:

```
MCR p15, 0, <Rt>, c9, c5, 0    ; fetch and lock instruction cache line,
                               ; Rt = MVA
MCR p15, 0, <Rt>, c9, c5, 1    ; unlock instruction cache,
                               ; Rt ignored
MCR p15, 0, <Rt>, c9, c6, 0    ; write Format D Data Cache Lockdown Register,
                               ; Rt = set or clear lockdown mode
MRC p15, 0, <Rt>, c9, c6, 0    ; read Format D Data Cache Lockdown Register,
                               ; Rt = lockdown mode status
MCR p15, 0, <Rt>, c9, c6, 1    ; unlock data cache,
                               ; Rt ignored
```

Note

Some format D implementations use CRm == {c1, c2} instead of CRm == {c5, c6}. You must check the Technical Reference Manual to find the encoding uses. The architecture did not require the implementation of CP15, and the Architecture Reference Manual only gave a recommended implementation. The actual CP15 implementation is IMPLEMENTATION DEFINED in ARMv4 and ARMv5.

The following rules determine how many entries in a cache set can be locked:

- At least one entry per cache set must be left for normal cache operation, otherwise behavior is UNPREDICTABLE.
- How many ways in each cache set can be locked is IMPLEMENTATION DEFINED. MAX_CACHESET_ENTRIES_LOCKED < NWAYS.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

7. キャッシュウェイ i 内でロックダウンされるキャッシュラインのそれぞれについて、次の処理を行います。
   - データキャッシュまたは統一キャッシュをロックダウンする場合は、LDR 命令を使用してメモリ キャッシュラインから 1 ワードをロードします。この操作により、メモリ キャッシュラインがキャッシュにロードされることが保証されます。
   - 命令キャッシュをロックダウンする場合は、CP15 c7 の命令キャッシュラインのプリフェッチ操作を使用して、メモリ キャッシュラインをキャッシュにフェッチします。

8. ビット i を L = 1、それ以外のビットはすべてこのルーチンを開始する前の値に復元し、CRm == 0 を指定して CP15 c9 レジスタへの書き込みを行います。

フォーマット C のキャッシュロック解除の手順

キャッシュのロックダウンされた部分をロック解除するには、各ビットに L == 0 を設定し、CP15 c9 レジスタへの書き込みを行います。

フォーマット D のロックダウン

このフォーマットでは、キャッシュウェイ方式を使用せず、個別の L1 キャッシュライン エントリをロックします。方式は、命令キャッシュとデータキャッシュとの場合で異なります。

CP15 c9 のフォーマット D のキャッシュロックダウン レジスタにアクセスするため使用される命令と操作は、次のとおりです。

```
MCR p15, 0, <Rt>, c9, c5, 0    ; fetch and lock instruction cache line,
                               ; Rt = MVA
MCR p15, 0, <Rt>, c9, c5, 1    ; unlock instruction cache,
                               ; Rt ignored
MCR p15, 0, <Rt>, c9, c6, 0    ; write Format D Data Cache Lockdown Register,
                               ; Rt = set or clear lockdown mode
MRC p15, 0, <Rt>, c9, c6, 0    ; read Format D Data Cache Lockdown Register,
                               ; Rt = lockdown mode status
MCR p15, 0, <Rt>, c9, c6, 1    ; unlock data cache,
                               ; Rt ignored
```

注

フォーマット D の一部の実装では、CRm == {c5, c6} の代わりに CRm == {c1, c2} が使用されます。テクニカル リファレンスマニュアルを参照して、エンコードの使用法について確認する必要があります。アーキテクチャでは CP15 の実装は必須ではなく、アーキテクチャ リファレンスマニュアルでは推奨される実装のみが示されています。ARMv4 および ARMv5 では、実際の CP15 実装は実装定義です。

次の規則により、キャッシュセット内でロックできるエントリ数が決定されます。

- 通常のキャッシュ操作用に、キャッシュセットごとに最低 1 つのエントリを残しておく必要があります。残されていない場合、動作は予測不能です。
- 各キャッシュセット内でロックできるウェイ数は実装定義ですが、MAX_CACHESET_ENTRIES_LOCKED < NWAYS です。
- フォーマット D の追加のエントリをロックする操作が、ロック解除されたエントリとして割り当てられるか、無視されるかは実装定義です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- Whether attempts to lock additional entries in Format D are allocated as an unlocked entry or ignored is IMPLEMENTATION DEFINED.

For the instruction cache, a fetch and lock operation fetches and locks individual cache lines. Each cache line is specified by its MVA. To lock code into the instruction cache, the following rules apply:

- The routine used to lock lines into the instruction cache must be executed from Non-cacheable memory.
- The code being locked into the instruction cache must be Cacheable.
- The instruction cache must be enabled and invalidated before locking down cache lines.

If these rules are not applied, results are UNPREDICTABLE. Entries must be unlocked using the global instruction cache unlock command.

Cache lines must be locked into the data cache by first setting a global lock control bit. Data cache linefills occurring while the global lock control bit is set are locked into the data cache. To lock data into the data cache, the following rules apply:

- The data being locked must not exist in the cache. Cache clean and invalidate operations might be necessary to meet this condition.
- The data to be locked must be Cacheable.
- The data cache must be enabled.

c9, Format D Data or unified Cache Lockdown Register, DCLR2

The format of the format D Data or unified Cache Lockdown Register is:

31	1	0
UNK/SBZ		L

L (bit [0]) Lock bit

0 no locking occurs

1 all data fills are locked while this bit is set.

Interaction with CP15 c7 operations

Cache lockdown only prevents the normal replacement strategy used on cache misses choosing to reallocate cache lines in the locked down region. CP15 c7 operations that invalidate, clean, or clean and invalidate cache contents affect locked down cache lines as normal. If invalidate operations are used, you must ensure that they do not use virtual addresses or cache set/way combinations that affect the locked down cache lines. Otherwise, if it is difficult to avoid affecting the locked down cache lines, repeat the cache lockdown procedure afterwards.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

命令キャッシュの場合、フェッチとロックの操作により、個別のキャッシュラインがフェッチされてロックされます。各キャッシュラインは MVA により指定されます。コードを命令キャッシュにロックする場合は、次の規則が適用されます。

- ラインを命令キャッシュにロックするため使用されるルーチンは、キャッシュ不可メモリから実行する必要があります。
- 命令キャッシュにロックされるコードは、キャッシュ可能な必要があります。
- 命令キャッシュは、キャッシュラインをロックダウンする前に稼動して無効化する必要があります。

これらの規則が適用されない場合、結果は予測不能です。エントリは、グローバルな命令キャッシュのロック解除コマンドを使用してロック解除する必要があります。

キャッシュラインは、最初にグローバルなロック制御ビットをセットして、データキャッシュにロックする必要があります。グローバルなロック制御ビットがセットされている間に発生するデータキャッシュのラインフィルは、データキャッシュにロックされます。データをデータキャッシュにロックする場合、次の規則が適用されます。

- ロックされるデータはキャッシュに存在しないことが必要です。この条件を満たすため、キャッシュのクリーニングと無効化の操作が必要な場合があります。
- ロックされるデータはキャッシュ可能な必要があります。
- データキャッシュが稼動している必要があります。

c9、フォーマット D のデータまたは統一キャッシュロックダウン レジスタ、DCLR2

フォーマット D のデータまたは統一キャッシュロックダウン レジスタのフォーマットは次のとおりです。

31	1	0
UNK/SBZ		L

L、ビット [0] ロックビット。

0 ロックは発生しません。

1 このビットがセットされている間、すべてのデータフィルがロックされます。

CP15 c7 操作との相互動作

キャッシュロックダウンでは、キャッシュミスで使用される通常の置換方式で、キャッシュラインをロックダウンされた領域に再割り当てする操作のみが回避されます。キャッシュ内容の無効化、クリーニング、またはクリーニングと無効化を行う CP15 c7 操作は、通常と同じにロックダウンされたキャッシュラインに影響を与えます。無効化操作を使用する場合は、その操作において、ロックダウンされたキャッシュラインに影響を与える仮想アドレスまたはキャッシュセット / ウェイの組み合わせが使用されていないことを確認する必要があります。ロックダウンされたキャッシュラインに影響を与えることを回避するのが困難な場合は、後でキャッシュロックダウンの手順を繰り返して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.7.12 c9, TCM support

TCM register support is optional when CP15 and TCM are supported in ARMv4 and ARMv5. For details see *c9, TCM support* on page AppxG-46.

H.7.13 c10, VMSA TLB lockdown support

TLB lockdown is an optional feature that enables the results of specified translation table walks to load into the TLB in a way that prevents them being overwritten by the results of subsequent translation table walks.

Translation table walks can take a long time because they involve potentially slow main memory accesses. In real-time interrupt handlers, translation table walks caused by the TLB that do not contain translations for the handler or the data it accesses can increase interrupt latency significantly.

Two basic lockdown models are supported:

- a TLB lock by entry model
- a translate and lock model introduced as an alternative model in ARMv5TE.

In an ARMv6 implementation that includes the Security Extensions, c10 TLB Lockdown registers are Configurable access registers, with access controlled by the NSACR. For more information, see:

- *Configurable access CP15 registers* on page B3-74 for general information
- *c1, Non-Secure Access Control Register (NSACR)* on page B3-110 and *c1, VMSA Security Extensions support* on page AppxG-35 for details of the NSACR.

From ARMv7, TLB lockdown is IMPLEMENTATION DEFINED with no recommended formats or mechanisms on how it is achieved other than reserved CP15 register space. See *TLB lockdown* on page B3-56 and *CP15 c10, Memory remapping and TLB control registers* on page B3-142.

Table H-25 shows the TLB operations used to support the different mechanisms.

Table H-25 TLB lockdown register support

Register or operation	Mechanism	CRn	opc1	CRm	opc2
Data or unified TLB Lockdown Register, DTLBLR	By entry	c10	0	c0[a]	0
Instruction TLB Lockdown Register, ITLBLR	By entry	c10	0	c0[a]	1
Lock instruction TLB	Translate and lock	c10	0	c4[b]	0
Unlock instruction TLB	Translate and lock	c10	0	c4[b]	1
Lock data TLB	Translate and lock	c10	0	c8[b]	0
Unlock data TLB	Translate and lock	c10	0	c8[b]	1

a. Read/write register that can be accessed using MCR and MRC instructions.
b. Write-only operation that is accessed only using the MCR instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

H.7.12 c9、TCM のサポート

ARMv4 および ARMv5 では、CP15 と TCM がサポートされている場合、TCM レジスタのサポートはオプションです。詳細については、*c9、TCM のサポート* P. 付録 G-46 を参照して下さい。

H.7.13 c10、VMSA TLB ロックダウンのサポート

TLB ロックダウンはオプション機能で、指定した変換テーブルウォークの結果を TLB にロードし、以後の変換テーブルウォークの結果で上書きされないようにできます。

潜在的に低速のメインメモリ アクセスが含まれているため、変換テーブルウォークには長時間を要する場合があります。リアルタイム割り込みハンドラでは、アクセスするハンドラまたはデータの変換が含まれていない TLB によって引き起こされる変換テーブルウォークにより、割り込みレイテンシが大幅に増加する場合があります。

2 つの基本的なロックダウンモデルがサポートされます。

- エントリモデルによる TLB ロック
- ARMv5TE で代替モデルとして導入された変換およびロックモデル

セキュリティ拡張機能が含まれている ARMv6 実装では、c10 TLB ロックダウンレジスタは構成可能アクセスレジスタで、アクセスは NSACR によって制御されます。詳細については、次のセクションを参照して下さい。

- *構成可能アクセスの CP15 レジスタ* P. B3-74（一般的な情報）
- *c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3-110 および *c1、VMSA セキュリティ拡張機能のサポート* P. 付録 G-35（NSACR に関する詳細情報）

ARMv7 およびそれ以降では、TLB ロックダウンは実装定義で、予約 CP15 レジスタ空間以外で実行される場合の推奨されるフォーマットや機構は存在しません。詳細については、*TLB ロックダウン* P. B3-56 および *CP15 c10、メモリ再マッピングおよび TLB 制御レジスタ* P. B3-142 を参照して下さい。

それぞれの機構をサポートするために使用される TLB 操作を、表 H-25 に示します。

表 H-25 TLB ロックダウンレジスタのサポート

レジスタまたは操作	機構	CRn	opc1	CRm	opc2
データまたは統一 TLB ロックダウンレジスタ、DTLBLR	エントリ別	c10	0	c0[a]	0
命令 TLB ロックダウンレジスタ、ITLBLR	エントリ別	c10	0	c0[a]	1
命令 TLB のロック	変換とロック	c10	0	c4[b]	0
命令 TLB のロック解除	変換とロック	c10	0	c4[b]	1
データ TLB のロック	変換とロック	c10	0	c8[b]	0
データ TLB のロック解除	変換とロック	c10	0	c8[b]	1

a. MCR および MRC 命令を使用してアクセスできる読み出し / 書き込みレジスタ
b. MCR 命令の使用でのみアクセスできる書き込み専用操作

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The TLB lock by entry model

When a new entry is written to the TLB as the result of a translation table walk following a TLB miss, the Victim field of the appropriate TLB Lockdown Register is incremented. When the value of the Victim field reaches the maximum number of TLB entries, the incremented Victim field wraps to the value of the Base field.

The architecture permits a modified form of this where the Base field is fixed as zero. It is particularly appropriate where an implementation provides dedicated lockable entries (unified or Harvard) as a separate resource from the general TLB provision. To determine which form of the locking model is provided, write the Base field with all bits nonzero, read it back and check whether it is a nonzero value.

TLB Lockdown Register format, for the lockdown by entry mechanism

The format of the CP15 register used for the lockdown by entry form is:

31 – 32-W[a]	31-W – 32-2W	31-2W – 1	0
Base	Victim	Reserved	P

a. $W = \log_2(n)$, rounded up to an integer if necessary, where n is the number of TLB entries.

If the implementation has separate instruction and data TLBs, there are two variants of this register, selected by the `<opc2>` field of the `MCR` or `MRC` instruction used to access the CP15 c10 register:

`<opc2> == 0` Selects the data TLB lockdown register.

`<opc2> == 1` Selects the instruction TLB lockdown register.

If the implementation has a unified TLB, only one variant of this register exists, and `<opc2>` must be zero.

CRm must be c0 for `MCR` and `MRC` instructions that access the CP15 c10 register.

Writing the appropriate TLB lockdown by entry register has the following effects:

- The victim field specifies which TLB entry is replaced by the translation table walk result generated by the next TLB miss.
- The Base field constrains the TLB replacement strategy to only use the TLB entries numbered from (Base) to ((number of TLB entries)-1), provided the victim field is already in that range.
- Any translation table walk results written to TLB entries while P == 1 are protected from being invalidated by the CP15 c8 invalidate entire TLB operations. Ones written while P == 0 are invalidated normally by these operations.

Note

If the number of TLB entries is not a power of two, writing a value to either the Base or Victim fields that is greater than or equal to the number of TLB entries has UNPREDICTABLE results.

Reading the appropriate TLB lockdown by entry register returns the last values written to the Base field and the P bit, and the number of the next TLB entry to be replaced in the victim field.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

エントリモデルによる TLB ロック

TLB ミスに続く変換テーブルウォークの結果として TLB に新規エントリが書き込まれるときには、該当する TLB ロックダウンレジスタのヴィクティムフィールドがインクリメントされます。ヴィクティムフィールドの値が TLB エントリの最大数に達すると、インクリメントされたヴィクティムフィールドはベースフィールドの値にラップアラウンドされます。

アーキテクチャでは、ベースフィールドの値が 0 に固定されている場合、この動作は変更された形式で許可されます。これは、実装でロック可能な専用のエントリ（統一またはハーバード）が、一般的な TLB 機構とは別のリソースとして提供されている場合に、特に該当します。どの形式のロックモデルが提供されているかを決定するには、ベースフィールドのすべてのビットに非 0 値を書き込んで読み返し、読み出されたのが非 0 値であるかどうかをチェックします。

TLB ロックダウンレジスタのフォーマット（エントリ機構によるロックダウンの場合）

エントリ形式によるロックダウンで使用される CP15 レジスタのフォーマットは次のとおりです。

31 – 32 − Wa	31 − W – 32 − 2W	31 − 2W – 1	0
ベース	ヴィクティム	予約	P

a. $W = \log_2(n)$。必要な場合はより大きな整数に丸められます。ここで、n は TLB エントリの数です。

実装に独立した命令とデータの TLB が含まれている場合、このレジスタには 2 つのバリアントが存在します。これは、CP15 c10 レジスタにアクセスするために使用される `MCR` または `MRC` 命令の `<opc2>` フィールドによって選択されます。

`<opc2> == 0` データ TLB ロックダウンレジスタが選択されます。

`<opc2> == 1` 命令 TLB ロックダウンレジスタが選択されます。

実装に統一 TLB が含まれている場合、このレジスタのバリアントは 1 つだけで、`<opc2>` は 0 にする必要があります。

CP15 c10 レジスタにアクセスする `MCR` および `MRC` 命令の場合は、CRm を c0 に設定する必要があります。

エントリレジスタにより該当する TLB ロックダウンを書き込むと、次の動作が行われます。

- ヴィクティムフィールドにより、次の TLB ミス発生時に生成される変換テーブルウォークの結果に置き換えられる TLB エントリが指定されます。
- ヴィクティムフィールドがすでに（ベース）～（(TLB エントリの数) − 1）の範囲にある場合、ベースフィールドによりその範囲の TLB エントリのみを使用するように TLB 置換方式が制限されます。
- P == 1 の間に TLB エントリに書き込まれる変換テーブルウォーク結果は、CP15 c8 の TLB 全体の無効化操作によって無効化されないように保護されます。P == 0 の間に書き込まれた結果は、通常はこれらの操作によって無効化されます。

注

TLB エントリの数が 2 のべき乗ではない場合、ベースまたはヴィクティムフィールドのいずれかに TLB エントリの数以上の値を書き込むと、結果は予測不能です。

エントリレジスタにより該当する TLB ロックダウンを読み出すと、ベースフィールドと P ビットに書き込まれた最後の値、およびヴィクティムフィールドで置き換えられる次の TLB エントリの番号が返されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996−1998, 2000, 2004−2008 ARM Limited. All rights reserved.

TLB lockdown procedure, using the by entry model

The normal procedure for locking down N TLB entries where the Base field can be modified is as follows:

1. Ensure that no processor exceptions can occur during the execution of this procedure, for example by disabling interrupts.
2. If an instruction TLB or unified TLB is being locked down, write the appropriate version of register c10 with Base == N, Victim == N, and P == 0. If appropriate, turn off facilities like branch prediction that make instruction prefetching harder to understand.
3. Invalidate the entire TLB to be locked down.
4. If an instruction TLB is being locked down, ensure that all TLB entries are loaded that relate to any instruction that could be prefetched by the rest of the lockdown procedure. Provided care is taken about where the lockdown procedure starts, one TLB entry can usually cover all of these. This means that the first instruction prefetch after the TLB is invalidated can do this job.

   If a data TLB is being locked down, ensure that all TLB entries are loaded that relate to any data accessed by the rest of the lockdown procedure, including any inline literals used by its code. Usually the best way to do this is to avoid using inline literals in the lockdown procedure, and to put all other data used by it in an area covered by a single TLB entry, and then to load one data item.

   If a unified TLB is being locked down, do both of the above.
5. For each of value of i from 0 to N-1:
   a. Write to the CP15 c10 register with Base == i, Victim == i, and P == 1.
   b. Force a translation table walk to occur for the area of memory whose translation table walk result is to be locked into TLB entry i as follows:
      - If a data TLB or unified TLB is being locked down, load an item of data from the area of memory.
      - If an instruction TLB is being locked down, use the CP15 c7 prefetch instruction cache line operation defined in Table H-21 on page AppxH-49 to prefetch an instruction from the area of memory.
6. Write to the CP15 c10 register with Base == N, Victim == N, and P == 0.

Note

If the FCSE is being used, care is required in step 5b because:

- If a data TLB or a unified TLB is being locked down, the address used for the load instruction is subject to modification by the FCSE.
- If an instruction TLB is being locked down, the address used for the CP15 c7 operation is being treated as data and so is not subject to modification by the FCSE.

To minimize the possible confusion caused by this, ARM recommends that the lockdown procedure:

- starts by disabling the FCSE, by setting the PID to zero

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

エントリモデルを使用した TLB ロックダウンの手順

ベースフィールドを変更できる場合に N 個の TLB エントリをロックダウンする通常の手順は、次のとおりです。

1. 割り込みを不可能にするなどの方法で、この手順の実行中にプロセッサの例外が発生しないことを保証します。
2. 命令 TLB または統一 TLB をロックダウンする場合は、ベース = N、ヴィクティム = N、P = 0 に設定して、該当するバージョンのレジスタ c10 への書き込みを行います。必要な場合は、命令プリフェッチを認識するのが困難になる分岐予測などの機構をオフにします。
3. ロックダウンされるすべての TLB を無効化します。
4. 命令 TLB をロックダウンする場合は、ロックダウンの残りの手順でプリフェッチされる可能性のある命令に関連するすべての TLB エントリがロードされていることを確認します。ロックダウン手順の開始時に注意を払えば、通常は 1 つの TLB エントリでこれらのすべてを処理することができます。これは、TLB 無効化後の最初の命令プリフェッチでこのジョブを実行できることを意味します。

   データ TLB をロックダウンする場合は、そのコードで使用されるインラインリテラルなど、ロックダウンの残りの手順でアクセスされるデータに関連するすべての TLB エントリがロードされていることを確認します。通常は、これを実行する最良の方法は、ロックダウン手順でインラインリテラルを使用することを回避し、使用されるデータのすべてを、単一の TLB エントリによって処理される領域に置き、1 つのデータ項目をロードすることです。

   統一 TLB をロックダウンする場合は、上で説明した手順の両方を実行します。
5. 0 ～ N − 1 の各 i 値について、次の処理を行います。
   a. ベース = i、ヴィクティム = i、P = 1 に設定して、CP15 c10 レジスタへ書き込みます。
   b. 変換テーブルウォークの結果が TLB エントリ i にロックされるメモリの領域で、次のようにして変換テーブルウォークを強制的に発生させます。
      - データ TLB または統一 TLB をロックダウンする場合は、メモリの領域から 1 つのデータ項目をロードします。
      - 命令 TLB をロックダウンする場合は、P. 付録 H-49 の表 H-21 で定義されている CP15 c7 の命令キャッシュラインのプリフェッチ操作を使用して、メモリの領域から 1 つの命令をプリフェッチします。
6. ベース = N、ヴィクティム = N、P = 0 に設定して、CP15 c10 レジスタへ書き込みます。

注

FCSE が使用されている場合は、次の理由から、ステップ 5b で注意が必要です。

- データ TLB または統一 TLB をロックダウンする場合、ロード命令で使用されるアドレスは FCSE による変更の対象となります。
- 命令 TLB をロックダウンする場合、CP15 c7 操作で使用されるアドレスはデータとして処理され、FCSE による変更の対象となりません。

このことにより発生する可能性のある混乱を最小限にするために、ロックダウン手順で次の処理を行うことをお勧めします。

- PID を 0 に設定し、FCSE を非稼動にして手順を開始する。
- 必要な場合は、該当する PID 値と使用される仮想アドレスの上位 7 ビットとで OR 演算を行うことにより、修飾仮想アドレスを生成する。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

- where appropriate, generates modified virtual addresses itself by ORing the appropriate PID value into the top 7 bits of the virtual addresses it uses.

Where the Base field is fixed at zero, the algorithm can be simplified as follows:

1. Ensure that no processor exceptions can occur during the execution of this procedure, for example by disabling interrupts.
2. If any current locked entries must be removed, an appropriate sequence of invalidate single entry operations is required.
3. Turn off branch prediction.
4. If an instruction TLB is being locked down, ensure that all TLB entries are loaded that relate to any instruction that could be prefetched by the rest of the lockdown procedure. Provided care is taken about where the lockdown procedure starts, one TLB entry can usually cover all of these. This means that the first instruction prefetch after the TLB is invalidated can do this job.

   If a data TLB is being locked down, ensure that all TLB entries are loaded that relate to any data accessed by the rest of the lockdown procedure, including any inline literals used by its code. Usually the best way to do this is to avoid using inline literals in the lockdown procedure, and to put all other data used by it in an area covered by a single TLB entry, and then to load one data item.

   If a unified TLB is being locked down, do both of the above.
5. For each value of i from 0 to N-1:
   a. Write to the CP15 c10 register with Base == 0, Victim == i, and P == 1.
   b. Force a translation table walk to occur for the area of memory whose translation table walk result is to be locked into TLB entry i as follows:
      - If a data TLB or unified TLB is being locked down, load an item of data from the area of memory.
      - If an instruction TLB is being locked down, use the CP15 c7 prefetch instruction cache line operation defined in Table H-21 on page AppxH-49 to cause an instruction to be prefetched from the area of memory.
6. Clear the appropriate lockdown register.

TLB unlock procedure, using the by entry model

To unlock the locked down portion of the TLB after it has been locked down using the above procedure:

1. Use CP15 c8 operations to invalidate each single entry that was locked down.
2. Write to the CP15 c10 register with Base == 0, Victim == 0, and P == 0.

Note

Step 1 ensures that P == 1 entries are not left in the TLB. If they are left in the TLB, the entire TLB invalidation step of a subsequent TLB lockdown procedure does not have the required effect.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ベースフィールドが 0 に固定されている場合、アルゴリズムは次のように単純化できます。

1. 割り込みを不可能にするなどの方法で、この手順の実行中にプロセッサの例外が発生しないことを保証します。
2. 現在ロックされているエントリを除去する必要がある場合は、単一エントリを無効化する適切な操作シーケンスが必要です。
3. 分岐予測をオフにします。
4. 命令 TLB をロックダウンする場合は、ロックダウンの残りの手順でプリフェッチされる可能性のある命令に関連するすべての TLB エントリがロードされていることを確認します。ロックダウン手順の開始時に注意を払えば、通常は 1 つの TLB エントリでこれらのすべてを処理することができます。これは、TLB 無効化後の最初の命令プリフェッチでこのジョブを実行できることを意味します。

   データ TLB をロックダウンする場合は、そのコードで使用されるインラインリテラルなど、ロックダウンの残りの手順でアクセスされるデータに関連するすべての TLB エントリがロードされていることを確認します。通常は、これを実行する最良の方法は、ロックダウン手順でインラインリテラルを使用することを回避し、使用されるデータのすべてを、単一の TLB エントリによって処理される領域に置き、1 つのデータ項目をロードすることです。

   統一 TLB をロックダウンする場合は、上で説明した手順の両方を実行します。
5. 0 ～ N－1 の各 i 値について、次の処理を行います。
   a. ベース = 0、ヴィクティム = i、P = 1 に設定して、CP15 c10 レジスタへ書き込みます。
   b. 変換テーブルウォークの結果が TLB エントリ i にロックされるメモリの領域で、次のようにして変換テーブルウォークを強制的に発生させます。
      - データ TLB または統一 TLB をロックダウンする場合は、メモリの領域から 1 つのデータ項目をロードします。
      - 命令 TLB をロックダウンする場合は、P. 付録 H-49 の表 H-21 で定義されている CP15 c7 の命令キャッシュラインのプリフェッチ操作を使用して、メモリの領域から 1 つの命令をプリフェッチします。
6. 該当するロックダウンレジスタをクリアします。

エントリモデルを使用した TLB ロック解除の手順

上で説明した手順を使用して TLB をロックダウンした後に、TLB のロックダウンされている部分をロック解除するには、次のようにします。

1. CP15 c8 操作を使用して、ロックダウンされているそれぞれの単一エントリを無効化します。
2. ベース = 0、ヴィクティム = 0、P = 0 に設定して、CP15 c10 レジスタへ書き込みます。

注

手順 1 により、P = 1 であるエントリが TLB に残っていないことが保証されます。TLB にエントリが残っていると、以後の TLB ロックダウン手順で行われる TLB 全体の無効化手順で、必要な効果が得られません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

The translate and lock model

This mechanism uses explicit TLB operations to translate and lock specific addresses into the TLB. Entries are unlocked on a global basis using the unlock operations. Addresses are loaded using their MVA. The following actions are UNPREDICTABLE:

- accessing these functions with read (MRC) commands
- using functions when the MMU is disabled
- trying to translate and lock an address that is already present in the TLB.

Any abort generated during the translation is reported as a lock abort in the FSR. Only external aborts and Translation faults are guaranteed to be detected. Any access permission, domain, or alignment checks on these functions are IMPLEMENTATION DEFINED. Operations that generate an abort do not affect the target TLB.

Where this model is applied to a unified TLB, the data TLB operations must be used.

Invalidate_all (I,D, or I and D) operations have no effect on locked entries.

TLB lockdown procedure, using the translate and lock model

All previously locked entries can be unlocked by issuing the appropriate unlock operation, I or D side. Explicit lockdown operations are then issued with the required MVA in register Rt.

TLB unlock procedure, using the translate and lock model

Issuing the appropriate unlock (I or D) TLB operation unlocks all locked entries. It is IMPLEMENTATION DEFINED whether an invalidate by MVA TLB operation removes the lock condition.

Note

The invalidate behavior is different in the TLB locking by entry model, where the invalidate by MVA operation is guaranteed to occur.

H.7.14 c13, VMSA FCSE support

The FCSE described in Appendix E *Fast Context Switch Extension (FCSE)* is an IMPLEMENTATION DEFINED option in ARMv4 and ARMv5. The feature is supported by the FCSEIDR as described in *c13, FCSE Process ID Register (FCSEIDR)* on page B3-152. The Context ID and Software Thread ID registers listed for ARMv7 are not supported in ARMv4 and ARMv5.

H.7.15 c15, IMPLEMENTATION DEFINED

CP15 c15 is reserved for IMPLEMENTATION DEFINED use. It is typically used for processor-specific runtime and test features.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

変換とロックモデル

この機構では、明示的な TLB 操作を使用して、特定のアドレスを変換して TLB にロックします。エントリは、ロック解除操作を使用してグローバルにロック解除されます。アドレスは、MVA を使用してロードされます。次のアクションは予測不能です。

- 読み出し (MRC) コマンドを使用して、これらの機能へアクセスする。
- MMU が非稼動の場合に機能を使用する。
- すでに TLB 内に存在するアドレスの変換とロックを試みる。

変換時に生成されたアボートは、FSR でロックアボートとして報告されます。外部アボートと変換フォールトのみが確実に検出されます。これらの機能でのアクセス許可、ドメイン、アライメントチェックは実装定義です。アボートを生成する操作は、ターゲット TLB には影響を与えません。

このモデルを統一 TLB に適用する場合、データ TLB の操作を使用する必要があります。

Invalidate_all（I、D、または I および D）操作は、ロックされているエントリには影響を与えません。

変換とロックモデルを使用した TLB ロックダウンの手順

以前にロックされたエントリはすべて、適切なロック解除操作（I または D 側）を発行してロック解除できます。その後で明示的なロックダウン操作が、レジスタ Rt の必須 MVA を使用して発行されます。

変換とロックモデルを使用した TLB ロック解除の手順

適切なロック解除（I または D）TLB 操作を発行すると、ロックされているすべてのエントリがロック解除されます。MVA による TLB 無効化操作によりロック状態が除去されるかどうかは実装定義です。

注

無効化動作はエントリモデルによる TLB ロックでは異なる動作で、この場合 MVA による無効化操作が確実に発生します。

H.7.14 c13、VMSA FCSE のサポート

ARMv4 および ARMv5 では、付録 E *「高速コンテキストスイッチ拡張機能 (FCSE)」* で説明されている FCSE は実装定義オプションです。この機能は、*c13、FCSE プロセス ID レジスタ (FCSEIDR)* P. B3- 152 で説明されているように FCSEIDR によってサポートされます。ARMv7 用にリストされているコンテキスト ID およびソフトウェアスレッド ID レジスタは、ARMv4 および ARMv5 ではサポートされません。

H.7.15 C15、実装定義

CP15 c15 は実装定義の使用に予約されています。通常、これはプロセッサ固有の実行時およびテスト機能で使用されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Appendix I
Pseudocode Definition

This appendix provides a definition of the pseudocode used in this manual, and lists the *helper* procedures and functions used by pseudocode to perform useful architecture-specific jobs. It contains the following sections:

- *Instruction encoding diagrams and pseudocode* on page AppxI-2
- *Limitations of pseudocode* on page AppxI-4
- *Data types* on page AppxI-5
- *Expressions* on page AppxI-9
- *Operators and built-in functions* on page AppxI-11
- *Statements and program structure* on page AppxI-17
- *Miscellaneous helper procedures and functions* on page AppxI-22.

Note

The pseudocode in this manual describes ARMv7. Where it can reasonably also describe the differences in earlier versions of the architecture, it does so. However, it does not always do so. For details of the differences in earlier architectures, see Appendix G *ARMv6 Differences* and Appendix H *ARMv4 and ARMv5 Differences*.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

付録 I
擬似コードの定義

この付録では、本書で使用されている擬似コードの定義を示します。また、擬似コードによって使用される、有用なアーキテクチャ固有のジョブを実行するためのヘルパプロシージャおよび関数のリストも示します。本章は次のセクションから構成されています。

- *命令エンコード図および擬似コード* P. 付録 I-2
- *擬似コードの制限* P. 付録 I-4
- *データ型* P. 付録 I-5
- *式* P. 付録 I-9
- *演算子と組込み関数* P. 付録 I-11
- *文とプログラム構造* P. 付録 I-17
- *その他のヘルパプロシージャおよび関数* P. 付録 I-22

注

本書では、ARMv7 の擬似コードについて説明します。アーキテクチャの以前のバージョンでの相違点についても説明することが妥当である場合は、その相違点についても説明されています。ただし、常に説明があるとは限りません。以前のアーキテクチャでの相違点の詳細については、付録 G「*ARMv6 との相違点*」および付録 H「*ARMv4 および ARMv5 での相違点*」を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.1 Instruction encoding diagrams and pseudocode

Instruction descriptions in this manual contain:

- An Encoding section, containing one or more encoding diagrams, each followed by some encoding-specific pseudocode that translates the fields of the encoding into inputs for the common pseudocode of the instruction, and picks out any encoding-specific special cases.
- An Operation section, containing common pseudocode that applies to all of the encodings being described. The Operation section pseudocode contains a call to the `EncodingSpecificOperations()` function, either at its start or after only a condition check performed by `if ConditionPassed() then`.

An encoding diagram specifies each bit of the instruction as one of the following:

- An obligatory 0 or 1, represented in the diagram as 0 or 1. If this bit does not have this value, the encoding corresponds to a different instruction.
- A *should be* 0 or 1, represented in the diagram as (0) or (1). If this bit does not have this value, the instruction is UNPREDICTABLE.
- A named single bit or a bit in a named multi-bit field. The `cond` field in bits [31:28] of many ARM instructions has some special rules associated with it.

An encoding diagram matches an instruction if all obligatory bits are identical in the encoding diagram and the instruction, and one of the following is true:

- the encoding diagram is not for an ARM instruction
- the encoding diagram is for an ARM instruction that does not have a `cond` field in bits [31:28]
- the encoding diagram is for an ARM instruction that has a `cond` field in bits [31:28], and bits [31:28] of the instruction are not 0b1111.

The execution model for an instruction is:

1. Find all encoding diagrams that match the instruction. It is possible that no encoding diagrams match. In that case, abandon this execution model and consult the relevant instruction set chapter instead to find out how the instruction is to be treated. The bit pattern of such an instruction is usually reserved and UNDEFINED, though there are some other possibilities. For example, unallocated hint instructions are documented as being reserved and to be executed as NOPs.
2. If the operation pseudocode for the matching encoding diagrams starts with a condition check, perform that condition check. If the condition check fails, abandon this execution model and treat the instruction as a NOP. If there are multiple matching encoding diagrams, either all or none of their corresponding pieces of common pseudocode start with a condition check.
3. Perform the encoding-specific pseudocode for each of the matching encoding diagrams independently and in parallel. Each such piece of encoding-specific pseudocode starts with a bitstring variable for each named bit or multi-bit field in its corresponding encoding diagram, named the same as the bit or multi-bit field and initialized with the values of the corresponding bit(s) from the bit pattern of the instruction.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

I.1 命令エンコード図および擬似コード

本書における命令の説明には次のものが含まれています。

- エンコードセクションには 1 つ以上のエンコード図が含まれ、それぞれの図に続いて、エンコードのフィールドを命令の共通擬似コードの入力に変換するエンコード固有の擬似コードがあり、そのエンコード固有の特別なケースが示されています。
- 動作セクションには、説明されているエンコードすべてに適用される共通の擬似コードが含まれています。動作セクションの擬似コードには、開始時、または `if ConditionPassed() then` により条件チェックのみが実行された直後に、`EncodingSpecificOperations()` 関数への呼び出しが含まれます。

エンコード図では、命令の各ビットが次のいずれかとして指定されます。

- 強制的な 0 または 1 は、図の中では 0 または 1 として表されます。このビットがこの値を持たない場合、エンコードは別の命令に対応します。
- 0 または 1 の*必要があるビット*。図の中では (0) または (1) として表されます。このビットがこの値を持たない場合、命令は予測不能です。
- 名前付きのビット、または名前付きの複数ビットフィールドに含まれている 1 ビット。多くの ARM 命令において、ビット [31:28] の `cond` フィールドには、それに関連する特別な規則が存在する場合があります。

必ず 0 または 1 の必要があるすべてのビットがエンコード図と命令とで一致する場合、そのエンコード図はその命令と一致し、次のいずれかが真になります。

- エンコード図は ARM 命令のものではない。
- エンコード図は、ビット [31:28] に `cond` フィールドを持たない ARM 命令のものである。
- エンコード図は、ビット [31:28] に `cond` フィールドを持つ ARM 命令のものであり、命令のビット [31:28] は 0b1111 ではない。

命令の実行モデルは次のとおりです。

1. その命令と一致するすべてのエンコード図を見つけます。一致するエンコード図が無いこともあります。その場合、その実行モデルを破棄して、代わりに、どのように命令が取り扱われるかを調べるために関連する命令セットの章を参照して下さい。その他の可能性もありますが、通常はそのような命令のビットパターンは、予約されていて未定義です。例えば、未割り当てのヒント命令は、予約されていて NOP として実行されると記載されています。
2. 一致するエンコード図の動作擬似コードが条件チェックから始まる場合は、その条件チェックを実行します。条件チェックが失敗した場合は、その実行モデルを破棄して、命令を NOP として取り扱います。一致するエンコード図が複数存在する場合、それらに対応する共通擬似コードのすべてが条件チェックから開始されるか、またはいずれの擬似コードも開始されません。
3. 一致する各エンコード図について、個別に並行して、エンコード固有の擬似コードを実行します。これらのエンコード固有の擬似コードは、対応するエンコード図のそれぞれの名前付きビットまたは複数ビットフィールドのビットストリング変数で始まります。これらの変数には、ビットまたは複数ビットフィールドと同じ名前が付けられており、命令のビットパターンに含まれている対応するビットの値で初期化されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

In a few cases, the encoding diagram contains more than one bit or field with same name. In these cases, the values of all of those bits or fields must be identical. The encoding-specific pseudocode contains a special case using the `Consistent()` function to specify what happens if they are not identical. `Consistent()` returns `TRUE` if all instruction bits or fields with the same name as its argument have the same value, and `FALSE` otherwise.

If there are multiple matching encoding diagrams, all but one of the corresponding pieces of pseudocode must contain a special case that indicates that it does not apply. Discard the results of all such pieces of pseudocode and their corresponding encoding diagrams.

There is now one remaining piece of pseudocode and its corresponding encoding diagram left to consider. This pseudocode might also contain a special case, most commonly one indicating that it is UNPREDICTABLE. If so, abandon this execution model and treat the instruction according to the special case.

4. Check the *should be* bits of the encoding diagram against the corresponding bits of the bit pattern of the instruction. If any of them do not match, abandon this execution model and treat the instruction as UNPREDICTABLE.

5. Perform the rest of the operation pseudocode for the instruction description that contains the encoding diagram. That pseudocode starts with all variables set to the values they were left with by the encoding-specific pseudocode.

The `ConditionPassed()` call in the common pseudocode (if present) performs step 2, and the `EncodingSpecificOperations()` call performs steps 3 and 4.

I.1.1 Pseudocode

The pseudocode provides precise descriptions of what instructions do, subject to the limitations described in *Limitations of pseudocode* on page AppxI-4. Instruction fields are referred to by the names shown in the encoding diagram for the instruction. The pseudocode is described in detail in the sections:

- *Data types* on page AppxI-5
- *Expressions* on page AppxI-9
- *Operators and built-in functions* on page AppxI-11
- *Statements and program structure* on page AppxI-17

Some pseudocode helper functions are described in *Miscellaneous helper procedures and functions* on page AppxI-22.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

エンコード図に、同じ名前のビットまたはフィールドが複数含まれている場合もあります。これらの場合は、ビットまたはフィールドのすべての値が同一の必要があります。エンコード固有の擬似コードには、それらが同一でない場合に、`Consistent()` 関数を使用して発生する動作を指定するという特殊なケースが含まれます。命令に含まれている、引数と同じ名前のすべてのビットまたはフィールドが同じ値である場合、`Consistent()` は `TRUE` を返します。異なる値の場合は `FALSE` を返します。

一致するエンコード図が複数存在する場合、それらに対応する擬似コードのうち、1 つを除くすべてに、それが適用されないことを示す特殊なケースが含まれている必要があります。それらの擬似コードと、対応するエンコード図の結果をすべて破棄します。

これによって、1 つの擬似コードとそれに対応するエンコード図が考慮の対象として残ります。この擬似コードにも特殊なケースが含まれている可能性があります。ほとんどの場合、これが予測不能であることを示すものです。その場合は、この実行モデルを破棄して、命令をその特殊なケースに従って取り扱います。

4. エンコード図の *0 または 1 の必要があるビット*を、命令のビットパターンの対応するビットと照合します。一致しないビットがある場合は、この実行モデルを破棄して、命令を予測不能として取り扱います。

5. エンコード図を含む命令記述の残りの動作擬似コードを実行します。この擬似コードは、エンコード固有の擬似コードにより残された値に設定された、すべての変数を使用して開始されます。

共通の擬似コードでの `ConditionPassed()`（存在する場合）呼び出しは、手順 2 を実行します。`EncodingSpecificOperations()` 呼び出しは、手順 3 と 4 を実行します。

I.1.1 擬似コード

擬似コードは、命令が実行する動作を正確に記述したもので、*擬似コードの制限* P. 付録 I-4 で説明されている制限に従います。命令フィールドは、命令のエンコード図で示されている名前により参照されます。擬似コードについては、次に示すセクションで詳しく説明します。

- *データ型* P. 付録 I-5
- *式* P. 付録 I-9
- *演算子と組込み関数* P. 付録 I-11
- *文とプログラム構造* P. 付録 I-17

一部の擬似コードヘルパ関数については、*その他のヘルパプロシージャおよび関数* P. 付録 I-22 で説明されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.2 Limitations of pseudocode

The pseudocode descriptions of instruction functionality have a number of limitations. These are mainly due to the fact that, for clarity and brevity, the pseudocode is a sequential and mostly deterministic language.

These limitations include:

- Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses, except in the case of `SWP` and `SWPB` instructions where the two accesses are to the same memory location. For a description of the ordering requirements on memory accesses see *Memory access order* on page A3-41.
- Pseudocode does not describe the exact rules when an `UNDEFINED` instruction fails its condition check. In such cases, the `UNDEFINED` pseudocode statement lies inside the `if ConditionPassed() then ...` structure, either directly or in the `EncodingSpecificOperations()` function call, and so the pseudocode indicates that the instruction executes as a NOP. *Conditional execution of undefined instructions* on page B1-51 describes the exact rules.
- Pseudocode does not describe the exact ordering requirements when one VFP instruction generates more than one floating-point exception. The exact rules are described in *Combinations of exceptions* on page A2-44.
- The pseudocode statements `UNDEFINED`, `UNPREDICTABLE` and `SEE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:
  - Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
  - No subsequent behavior indicated by the pseudocode occurs. This means that these statements terminate pseudocode execution.

  For more information, see *Simple statements* on page AppxI-17.
- A processor exception can be taken during execution of the pseudocode for an instruction, either explicitly as a result of the execution of a pseudocode function such as `DataAbort()`, or implicitly, for example if an interrupt is taken during execution of an `LDM` instruction. If this happens, the pseudocode does not describe the extent to which the normal behavior of the instruction occurs. To determine that, see the descriptions of the processor exceptions in *Exceptions* on page B1-30.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

I.2 擬似コードの制限

命令の機能の擬似コードによる記述には、いくつか制限があります。これらの制限は主に、明瞭かつ簡潔にするために、擬似コードがシーケンシャルであいまいさのない言語であることが原因です。

制限には次のものが含まれます。

- 命令で複数のメモリアクセスが生成される場合、擬似コードでは順序付け要件については記述されません。ただし、同じメモリ位置に対して 2 つのアクセスが存在する `SWP` および `SWPB` 命令の場合は除きます。メモリアクセス時の順序付け要件の説明については、*メモリアクセスの順序* P. A3-41 を参照して下さい。
- 擬似コードでは、未定義の命令が条件チェックに失敗した場合の厳密なルールが記述されていません。そのような場合は、擬似コードの `UNDEFINED` ステートメントが、直接または `EncodingSpecificOperations()` 関数の呼び出し内で `if ConditionPassed() then ...` 構造の中に置かれるため、擬似コードでは、命令が NOP として実行されることが示されます。厳密な規則については、*未定義命令の条件付き実行* P. B1-51 で説明されています。
- 擬似コードでは、1 つの VFP 命令が複数の浮動小数点例外を生成する場合、厳密な順序付けの要件は記述されません。厳密な規則については、*例外の組み合わせ* P. A2-44 で説明されています。
- 擬似コードの `UNDEFINED`、`UNPREDICTABLE`、`SEE` ステートメントは、実行される擬似コードによって示されているのとは異なる動作を示しています。これらのステートメントのいずれかが実行されると、次の動作が行われます。
  - 擬似コードにより示されている、それ以前の動作は、そのステートメントが実行されたと判定するに必要な範囲まで動作が進んだものとしてのみ指定されます。
  - 擬似コードにより示されている以後の動作は実行されません。つまり、これらのステートメントによって擬似コードの実行は終了します。

  詳細については、*単純な文* P. 付録 I-17 を参照して下さい。
- プロセッサ例外は、命令の擬似コードの実行中に、`DataAbort()` などの擬似コード関数の実行の結果として明示的に、または例えば、`LDM` 命令の実行中に割り込みが取得された場合などに暗黙的に取得されることがあります。この例外が発生した場合、擬似コードでは、命令の通常の動作が行われる範囲については記述されません。これを特定するには、*例外* P. B1-30 にあるプロセッサ例外の説明を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.3 Data types

This section describes:

- *General data type rules*
- *Bitstrings*
- *Integers* on page AppxI-6
- *Reals* on page AppxI-6
- *Booleans* on page AppxI-6
- *Enumerations* on page AppxI-6
- *Lists* on page AppxI-7
- *Arrays* on page AppxI-8.

I.3.1 General data type rules

ARM architecture pseudocode is a strongly-typed language. Every constant and variable is of one of the following types:

- bitstring
- integer
- boolean
- real
- enumeration
- list
- array.

The type of a constant is determined by its syntax. The type of a variable is normally determined by assignment to the variable, with the variable being implicitly declared to be of the same type as whatever is assigned to it. For example, the assignments `x = 1`, `y = '1'`, and `z = TRUE` implicitly declare the variables `x`, `y` and `z` to have types integer, length-1 bitstring and boolean respectively.

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. This is most often done in function definitions for the arguments and the result of the function.

These data types are described in more detail in the following sections.

I.3.2 Bitstrings

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum permitted length of a bitstring is 1.

The type name for bitstrings of length `N` is `bits(N)`. A synonym of `bits(1)` is `bit`.

Bitstring constants are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants of type `bit` are `'0'` and `'1'`. Spaces can be included in bitstrings for clarity.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.3 データ型

このセクションでは、次のトピックについて説明します。

- *データ型の一般的な規則*
- *ビットストリング*
- *整数* P. 付録 I-6
- *実数* P. 付録 I-6
- *ブール* P. 付録 I-6
- *列挙型* P. 付録 I-6
- *リスト* P. 付録 I-7
- *配列* P. 付録 I-8

I.3.1 データ型の一般的な規則

ARM アーキテクチャの擬似コードは、強く型付けされた言語です。すべての定数と変数は、次のいずれかの型です。

- ビットストリング
- 整数
- ブール
- 実数
- 列挙型
- リスト
- 配列

定数の型はその構文により決定されます。通常、変数の型は、変数への代入により決定されます。これは、変数は代入されるデータと同じ型であることが暗黙的に宣言されているためです。例えば、代入 `x = 1`、`y = '1'`、および `z = TRUE` では、変数 `x`、`y`、`z` が、それぞれ整数、長さ 1 のビットストリング、およびブールの型であることを暗黙的に宣言していることになります。

型の名前を変数名の前に置くことにより、変数の型を明示的に宣言することもできます。これはほとんどの場合、関数の定義で引数や結果の型を宣言するために行われます。

これらのデータ型については、以下のセクションで詳しく説明します。

I.3.2 ビットストリング

ビットストリングは、0 および 1 のビットの、有限の長さのストリングです。長さが異なるビットストリングは、異なる型と見なされます。ビットストリングの最小の長さは 1 です。

長さ `N` のビットストリングの型名は `bits(N)` です。`bits(1)` は `bit` と同義です。

ビットストリング定数は、0 と 1 のストリングを一重引用符 (') で囲った形式で表記されます。例えば、`bit` 型の 2 つの定数は、`'0'` および `'1'` と表記されます。ビットストリングには、分かりやすくするためにスペースを含めることができます。

`'x'` 個のビットを持つ特別な形式のビットストリング定数は、ビットストリングの比較で許可されます。詳細については、*等価と非等価のテスト* P. 付録 I-11 を参照して下さい。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

A special form of bitstring constant with `'x'` bits is permitted in bitstring comparisons. For details see *Equality and non-equality testing* on page AppxI-11.

Every bitstring value has a left-to-right order, with the bits being numbered in standard *little-endian* order. That is, the leftmost bit of a bitstring of length N is bit N–1 and its right-most bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings derived from encoding diagrams, this order matches the way they are printed.

Bitstrings are the only concrete data type in pseudocode, in the sense that they correspond directly to the contents of registers, memory locations, instructions, and so on. All of the remaining data types are abstract.

I.3.3 Integers

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable functions to interpret those bitstrings as integers.

The type name for integers is `integer`.

Integer constants are normally written in decimal, such as `0`, `15`, `-1234`. They can also be written in C-style hexadecimal, such as `0x55` or `0x80000000`. Hexadecimal integer constants are treated as positive unless they have a preceding minus sign. For example, `0x80000000` is the integer $+2^{31}$. If -2^{31} needs to be written in hexadecimal, it must be written as `-0x80000000`.

I.3.4 Reals

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable functions to interpret those bitstrings as reals.

The type name for reals is `real`.

Real constants are written in decimal with a decimal point (so `0` is an integer constant, but `0.0` is a real constant).

I.3.5 Booleans

A boolean is a logical true or false value.

The type name for booleans is `boolean`. This is not the same type as `bit`, which is a length-1 bitstring. Boolean constants are `TRUE` and `FALSE`.

I.3.6 Enumerations

An enumeration is a defined set of symbolic constants, such as:

```
enumeration InstrSet {InstrSet_ARM, InstrSet_Thumb, InstrSet_Jazelle, InstrSet_ThumbEE};
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

すべてのビットストリングの値は左から右の順序で表現され、ビットは標準的な*リトルエンディアン*順序で番号付けされます。つまり、長さ N のビットストリングの一番左のビットはビット N-1 で、一番右のビットはビット 0 です。この順序は、整数への変換または整数からの変換で、最上位ビットから最下位ビットへの順序として使用されます。ビットストリング定数、およびエンコード図から派生するビットストリングの場合、この順序は印刷されている順序と一致します。

ビットストリングは、レジスタの内容、メモリの位置、命令などと直接対応するという点において、擬似コードにおける唯一の具体的なデータ型です。残りのデータ型はすべて抽象的です。

I.3.3 整数

擬似コードの整数にはサイズの制限がなく、正と負の両方が許可されます。そのため、これらは、コンピュータ言語やアーキテクチャにおける一般的な意味での整数ではなく、数学的な意味の整数です。コンピュータの整数は、擬似コードでは適切な長さのビットストリングとして表され、これらのビットストリングを整数として解釈するための適切な関数と関連付けられています。

整数の型名は `integer` です。

整数の定数は通常、`0`、`15`、`- 1234` などのように 10 進数で表記されます。`0x55` や `0x80000000` のように、C 形式の 16 進数で表記されることもあります。16 進整数の定数は、マイナスの符号が値の前に無い場合は正の値として扱われます。例えば、`0x80000000` は整数 $+2^{31}$ です。-2^{31} を 16 進数で表記する必要がある場合は、`- 0x80000000` と表記します。

I.3.4 実数

擬似コードの実数はサイズと精度に制限がありません。そのため、これらは数学的な実数であり、コンピュータの浮動小数点数とは異なります。コンピュータの浮動小数点数は、擬似コードでは適切な長さのビットストリングとして表され、これらのビットストリングを実数として解釈するための適切な関数と関連付けられています。

実数の型名は `real` です。

実数の定数は、小数点付きの 10 進数で表記されます（このため、`0` は整数の定数ですが、`0.0` は実数の定数です）。

I.3.5 ブール

ブールは論理的な TRUE または FALSE の値です。

ブールの型名は `boolean` です。これは、長さ 1 のビットストリングである `bit` とは異なる型です。ブールの定数は `TRUE` と `FALSE` です。

I.3.6 列挙型

列挙型は、次のようなシンボリックな定数の定義されたセットです。

```
enumeration InstrSet {InstrSet_ARM, InstrSet_Thumb, InstrSet_Jazelle, InstrSet_ThumbEE};
```

列挙型は常に、1 つ以上のシンボリック定数を含みます。シンボリック定数を複数の列挙型で共有することはできません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

An enumeration always contains at least one symbolic constant, and symbolic constants are not permitted to be shared between enumerations.

Enumerations must be declared explicitly, though a variable of an enumeration type can be declared implicitly as usual by assigning one of the symbolic constants to it. By convention, each of the symbolic constants starts with the name of the enumeration followed by an underscore. The name of the enumeration is its type name, and the symbolic constants are its possible constants.

Note

Booleans are basically a pre-declared enumeration:

```
enumeration boolean {FALSE, TRUE};
```

that does not follow the normal naming convention and that has a special role in some pseudocode constructs, such as `if` statements.

I.3.7 Lists

A list is an ordered set of other data items, separated by commas and enclosed in parentheses, such as:

```
(bits(32) shifter_result, bit shifter_carry_out)
```

A list always contains at least one data item.

Lists are often used as the return type for a function that returns multiple results. For example, this particular list is the return type of the function `Shift_C()` that performs a standard ARM shift or rotation, when its first operand is of type `bits(32)`.

Some specific pseudocode operators use lists surrounded by other forms of bracketing than parentheses. These are:

- Bitstring extraction operators, that use lists of bit numbers or ranges of bit numbers surrounded by angle brackets "<…>".
- Array indexing, that uses lists of array indexes surrounded by square brackets "[…]".
- Array-like function argument passing, that uses lists of function arguments surrounded by square brackets "[…]".

Each combination of data types in a list is a separate type, with type name given by just listing the data types (that is, `(bits(32),bit)` in the above example). The general principle that types can be declared by assignment extends to the types of the individual list items in a list. For example:

```
(shift_t, shift_n) = ('00', 0);
```

implicitly declares `shift_t`, `shift_n` and `(shift_t,shift_n)` to be of types `bits(2)`, `integer` and `(bits(2),integer)` respectively.

A list type can also be explicitly named, with explicitly named elements in the list. For example:

```
type ShiftSpec is (bits(2) shift, integer amount);
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

列挙型の変数はシンボリック定数の 1 つを割り当てることにより、通常どおり暗黙的に宣言することが可能ですが、列挙型自体は明示的に宣言されている必要があります。慣例により、それぞれのシンボリック定数は、列挙型の名前から始まり、その後に下線が続きます。列挙型の名前は型名で、シンボリック定数はその定数になる可能性のあるものです。

注

ブールは基本的には、事前に宣言された列挙型です。

```
enumeration boolean {FALSE, TRUE};
```

この宣言は通常の命名規則に従っておらず、`if` ステートメントなどのいくつかの擬似コード構造で特別な役割を持っています。

I.3.7 リスト

リストは、他のデータ項目の順序付けされたセットです。これは、次のようにカンマで区切られ、括弧で囲まれています。

```
(bits(32) shifter_result, bit shifter_carry_out)
```

リストは常に、1 つ以上のデータ項目を含んでいます。

リストは、複数の結果を返す関数の戻り型として多く利用されます。例えば、この例のリストは、最初のオペランドの型が `bits(32)` のときに、標準的な ARM シフトまたはローテートを実行する、`Shift_C()` 関数の戻り型です。

いくつかの特定の擬似コードの演算子は、通常の括弧ではなく別形式の括弧で囲まれたリストを使用します。具体的には、次のものがあります。

- ビットストリング抽出演算子。これは不等号「<...>」で囲まれたビット番号またはビット番号の範囲のリストを使用します。
- 配列のインデクス付け。これは角括弧「[...]」で囲まれた配列インデクスのリストを使用します。
- 関数に配列と似た引数を渡す場合。これは角括弧「[...]」で囲まれた関数の引数のリストを使用します。

リストに含まれているデータ型のそれぞれの組み合わせは個別の型で、型名は単にデータ型をリストするだけで与えられます（上の例の場合、`(bits(32) ,bit)` です）。一般的な原則として、型は代入により宣言することができますが、この原則はリストの中の個別のリスト項目の型にまで広げることができます。次に例を示します。

```
(shift_t, shift_n) = ('00', 0);
```

上の式は、`shift_t`、`shift_n`、および `(shift_t,shift_n)` が、それぞれ `bits(2)`、`integer`、`(bits(2),integer)` 型であることを暗黙的に宣言しています。

リストの型は、明示的に名前が付けられた要素をリスト中で使って、明示的に指定することもできます。次に例を示します。

```
type ShiftSpec is (bits(2) shift, integer amount);
```

この定義と宣言の後に次のように記述します。

```
ShiftSpec abc;
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

After this definition and the declaration:

```
ShiftSpec abc;
```

the elements of the resulting list can then be referred to as "abc.shift" and "abc.amount". This sort of qualified naming of list elements is only permitted for variables that have been explicitly declared, not for those that have been declared by assignment only.

Explicitly naming a type does not alter what type it is. For example, after the above definition of ShiftSpec, ShiftSpec and (bits(2),integer) are two different names for the same type, not the names of two different types. To avoid ambiguity in references to list elements, it is an error to declare a list variable multiple times using different names of its type or to qualify it with list element names not associated with the name by which it was declared.

An item in a list that is being assigned to can be written as "-" to indicate that the corresponding item of the assigned list value is discarded. For example:

```
(shifted, -) = LSL_C(operand, amount);
```

List constants are written as a list of constants of the appropriate types, like ('00', 0) in the above example.

I.3.8 Arrays

Pseudocode arrays are indexed by either enumerations or integer ranges (represented by the lower inclusive end of the range, then "..", then the upper inclusive end of the range). For example:

```
enumeration PhysReg {
    PhysReg_R0,    PhysReg_R1,     PhysReg_R2,    PhysReg_R3,
    PhysReg_R4,    PhysReg_R5,     PhysReg_R6,    PhysReg_R7,
    PhysReg_R8,    PhysReg_R8fiq,  PhysReg_R9,    PhysReg_R9fiq,
    PhysReg_R10,   PhysReg_R10fiq, PhysReg_R11,   PhysReg_R11fiq,
    PhysReg_R12,   PhysReg_R12fiq,
    PhysReg_SP,    PhysReg_SPfiq,  PhysReg_SPirq, PhysReg_SPsvc, PhysReg_SPabt,
    PhysReg_SPund, PhysReg_SPmon,
    PhysReg_LR,    PhysReg_LRfiq,  PhysReg_LRirq, PhysReg_LRsvc, PhysReg_LRabt,
    PhysReg_LRund, PhysReg_LRmon,
    PhysReg_PC};
array bits(32) _R[PhysReg];
array bits(8) _Memory[0..0xFFFFFFFF];
```

Arrays are always explicitly declared, and there is no notation for a constant array. Arrays always contain at least one element, because enumerations always contain at least one symbolic constant and integer ranges always contain at least one integer.

Arrays do not usually appear directly in pseudocode. The items that syntactically look like arrays in pseudocode are usually array-like functions such as R[i], MemU[address,size] or Elem[vector,i,size]. These functions package up and abstract additional operations normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and Advanced SIMD element processing.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

結果として作成されるリストの要素は、"abc.shift" および "abc.amount" として参照することができます。このようなリストの要素の修飾名は、明示的に宣言された変数の場合のみ許可されており、代入でのみ宣言されたものでは許可されていません。

型に明示的に名前付けしても、型自体は変更されません。例えば、上記の ShiftSpec 宣言の後で、ShiftSpec および (bits(2) ,integer) は、同じ型を表す 2 つの異なる名前であり、2 つの異なる型の名前ではありません。リストの要素への参照が不明瞭になることを避けるために、型の異なる名前を使用してリスト変数を複数回数宣言したり、宣言に使用された名前と関連していないリストの要素名でリスト変数を修飾したりすると、エラーが発生します。

代入されるリストの項目は、代入されるリスト値の対応する項目が破棄されることを示すために、"-" と記述されることがあります。次に例を示します。

```
(shifted, -) = LSL_C(operand, amount);
```

リスト定数は、上の例にある ('00', 0) のように、適切な型の定数のリストとして記述されます。

I.3.8 配列

擬似コードの配列は、列挙型または整数の範囲（範囲の下限、".."、および範囲の上限により表されます。この範囲には、上限と下限の数値が含まれます）によりインデクス付けされます。次に例を示します。

```
enumeration PhysReg {
    PhysReg_R0,    PhysReg_R1,     PhysReg_R2,    PhysReg_R3,
    PhysReg_R4,    PhysReg_R5,     PhysReg_R6,    PhysReg_R7,
    PhysReg_R8,    PhysReg_R8fiq,  PhysReg_R9,    PhysReg_R9fiq,
    PhysReg_R10,   PhysReg_R10fiq, PhysReg_R11,   PhysReg_R11fiq,
    PhysReg_R12,   PhysReg_R12fiq,
    PhysReg_SP,    PhysReg_SPfiq,  PhysReg_SPirq, PhysReg_SPsvc, PhysReg_SPabt,
    PhysReg_SPund, PhysReg_SPmon,
    PhysReg_LR,    PhysReg_LRfiq,  PhysReg_LRirq, PhysReg_LRsvc, PhysReg_LRabt,
    PhysReg_LRund, PhysReg_LRmon,
    PhysReg_PC};
array bits(32) _R[PhysReg];
array bits(8) _Memory[0..0xFFFFFFFF];
```

配列は常に明示的に宣言され、定数配列の表記はありません。配列には常に 1 つ以上の要素が含まれます。これは、列挙型は常に 1 つ以上のシンボリック定数を含み、整数の範囲は常に 1 つ以上の整数を含んでいるためです。

通常、配列は擬似コードには直接出現しません。擬似コードで構文的に配列のように見える項目は、通常は R[i]、MemU[address,size]、Elem[vector,i,size] などの配列様関数です。これらの関数は、レジスタのバンク処理、メモリ保護、エンディアン依存のバイト順序、排他アクセスの保守、アドバンスト SIMD エレメントの処理など、通常は基礎となる配列へのアクセスに対して実行される追加操作をまとめて抽象化したものです。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.4 Expressions

This section describes:

- *General expression syntax*
- *Operators and functions - polymorphism and prototypes* on page AppxI-10
- *Precedence rules* on page AppxI-10.

I.4.1 General expression syntax

An expression is one of the following:

- a constant
- a variable, optionally preceded by a data type name to declare its type
- the word `UNKNOWN` preceded by a data type name to declare its type
- the result of applying a language-defined operator to other expressions
- the result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register described in the text is to be regarded as declaring a correspondingly named bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is defined as RAZ/WI, then the corresponding bit of its variable reads as 0 and ignore writes.

An expression like `bits(32) UNKNOWN` indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not constitute a security hole and must not be promoted as providing any useful information to software. (This was called an UNPREDICTABLE value in previous ARM architecture documentation. It is related to but not the same as UNPREDICTABLE, which says that the entire architectural state becomes similarly unspecified.)

A subset of expressions are assignable. That is, they can be placed on the left-hand side of an assignment. This subset consists of:

- Variables
- The results of applying some operators to other expressions. The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. (For example, those circumstances might include one or more of the expressions the operator operates on themselves being assignable expressions.)
- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.

Every expression has a data type. This is determined by:

- For a constant, the syntax of the constant.
- For a variable, there are three possible sources for the type
  - — its optional preceding data type name

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.4 式

このセクションでは、次のトピックについて説明します。

- *一般的な式の構文*
- *演算子と関数 - 多相性とプロトタイプ* P. 付録 I- 10
- *優先順位の規則* P. 付録 I- 10

I.4.1 一般的な式の構文

式は、次のいずれか 1 つの形式です。

- 定数
- 変数。オプションとして、型を宣言するデータ型名が前に記述されることがあります。
- 型を宣言するデータ型名の後にある、`UNKNOWN` という単語
- 言語で定義されている演算子を他の式に適用した結果
- 関数を他の式に適用した結果

変数名は通常、英数字と下線で構成され、最初の文字は英字または下線です。

テキストに記述されている各レジスタは、対応した名前を持つビットストリング変数の宣言とみなされ、その変数はそのレジスタに指定されている動作を行います。例えば、レジスタのあるビットが RAZ/WI と定義されている場合、その変数の対応するビットは 0 として読み出され、書き込みは無視されます。

`bits(32) UNKNOWN` のような式は、式の結果が指定された型の値であるが、アーキテクチャでは値が何であるかは指定されていないため、ソフトウェアはこのような値に依存してはならないことを示しています。生成された値がセキュリティホールを構成しないようにする必要があります。また、この値がソフトウェアに有用な情報を提供するものと見なさないようにすることも必要です（以前の ARM アーキテクチャのドキュメントでは、このような値は予測不能と表記されています。これは、アーキテクチャ全体の状態が同じように未指定状態になることを示す予測不能と関連していますが、同じではありません）。

式のサブセットは代入可能です。このため、これらは代入式の左側に置くことができます。このサブセットは、次のもので構成されます。

- 変数
- いくつかの演算子を他の式に適用した結果。代入可能な式を生成可能な言語定義の演算子の記述には、どのような条件で生成が行われるかが指定されています（例えば、演算子が、それ自身が代入可能な式に対して演算を行う、1 つ以上の式が条件に含まれる場合があります）。
- 配列様関数を他の式に適用した結果。配列様関数の記述には、代入可能な式を生成可能な条件が指定されています。

すべての式にはデータ型があります。式の型は、次のように決定されます。

- 定数の場合は、定数の構文
- 変数の場合は、次の 3 つが型のソースとなる可能性があります。
  - オプションの、前に記述されているデータ型名
  - この規則の再帰的な適用により、擬似コードで前もって与えられるデータ型

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

— a data type it was given earlier in the pseudocode by recursive application of this rule

— a data type it is being given by assignment (either by direct assignment to it, or by assignment to a list of which it is a member).

It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.

- For a language-defined operator, the definition of the operator.
- For a function, the definition of the function.

I.4.2 Operators and functions - polymorphism and prototypes

Operators and functions in pseudocode can be polymorphic, producing different functionality when applied to different data types. Each of the resulting forms of an operator or function has a different prototype definition. For example, the operator + has forms that act on various combinations of integers, reals and bitstrings.

One particularly common form of polymorphism is between bitstrings of different lengths. This is represented by using `bits(N)`, `bits(M)`, and so on, in the prototype definition.

I.4.3 Precedence rules

The precedence rules for expressions are:

1. Constants, variables and function invocations are evaluated with higher priority than any operators using their results.
2. Expressions on integers follow the normal *exponentiation before multiply/divide before add/subtract* operator precedence rules, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but need not be if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j` and `k` are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

– 代入により与えられるデータ型（直接代入、またはその変数がメンバであるリストへの代入のいずれかによる）

ある変数について、これら 3 つのデータ型のソースがいずれも存在しない場合、または複数のデータ型のソースが存在し、それらの型が一致しない場合、擬似コードはエラーとなります。

- 言語定義演算子の場合は、演算子の定義
- 関数の場合は、関数の定義

I.4.2 演算子と関数 – 多相性とプロトタイプ

擬似コードの演算子と関数は多相型にすることが可能で、適用されるデータ型によって異なる機能を持つようにできます。結果として生成される演算子または関数のそれぞれの形式は、異なるプロトタイプ定義を持っています。例えば、演算子 + には、整数、実数、ビットストリングスの各種の組み合わせについて機能する形式があります。

多相性のなかでも特によく使われる形式は、長さが異なる複数のビットストリングに対してのものです。これは、プロトタイプ定義で `bits(N)` や `bits(M)` などを使うことで表されます。

I.4.3 優先順位の規則

式の優先順位は次の規則で決定されます。

1. 定数、変数、および関数呼び出しは、それらの結果を使用しているどの演算子よりも高い優先順位で評価されます。
2. 整数の式は、演算子の通常の規則に従い、*累乗*、*乗算／除算*、*加算／減算*の順に計算されます。また、連続する乗算 / 除算または加算 / 減算は、左から右の順序で計算されます。
3. その他の式で、順序があいまいになる場合は、優先される演算子を示すために括弧で囲む必要があります。ただし、型の規則に従って使用可能な優先順序がすべて必然的に同じ結果を導く場合、その必要はありません。例えば、`i`、`j`、および `k` が整数の変数の場合、`i > 0 && j > 0 && k > 0` は許容されますが、`i > 0 && j > 0 || k > 0` は許容されません。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.5 Operators and built-in functions

This section describes:

- *Operations on generic types*
- *Operations on booleans*
- *Bitstring manipulation* on page AppxI-12
- *Arithmetic* on page AppxI-14.

I.5.1 Operations on generic types

The following operations are defined for all types.

Equality and non-equality testing

Any two values x and y of the same type can be tested for equality by the expression `x == y` and for non-equality by the expression `x != y`. In both cases, the result is of type `boolean`.

A special form of comparison with a bitstring constant that includes `'x'` bits as well as `'0'` and `'1'` bits is permitted. The bits corresponding to the `'x'` bits are ignored in determining the result of the comparison. For example, if opcode is a 4-bit bitstring, `opcode == '1x0x'` is equivalent to `opcode<3> == '1' && opcode<1> == '0'`. This special form is also permitted in the implied equality comparisons in `when` parts of `case ... of ...` structures.

Conditional selection

If x and y are two values of the same type and `t` is a value of type `boolean`, then `if t then x else y` is an expression of the same type as x and y that produces x if `t` is `TRUE` and y if `t` is `FALSE`.

I.5.2 Operations on booleans

If x is a `boolean`, then `!x` is its logical inverse.

If x and y are booleans, then `x && y` is the result of ANDing them together. As in the C language, if x is `FALSE`, the result is determined to be `FALSE` without evaluating y.

If x and y are booleans, then `x || y` is the result of ORing them together. As in the C language, if x is `TRUE`, the result is determined to be `TRUE` without evaluating y.

If x and y are booleans, then `x ^ y` is the result of exclusive-ORing them together.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.5 演算子と組込み関数

このセクションでは、次のトピックについて説明します。

- *すべての型に対して使用できる演算*
- *ブールの演算*
- *ビットストリング操作* P. 付録 I-12
- *算術演算* P. 付録 I-14

I.5.1 すべての型に対して使用できる演算

次の演算は、すべての型に対して定義されています。

等価と非等価のテスト

同じ型の 2 つの値 x および y が等価であることを式 `x == y` を使用して、等価でないことを式 `x != y` を使用してテストできます。いずれの場合も、結果は `boolean` 型です。

`'x'` ビット、`'0'` ビット、および `'1'` ビットを含むビットストリング定数を使った、特別な形式の比較を使用できます。`'x'` ビットと対応するビットは、比較の結果を決定するときに無視されます。例えば、オペコードが 4 ビットのビットストリングの場合、`opcode == '1x0x'` は `opcode<3> == '1' && opcode<1> == '0'` と等価です。この特殊な形式は、`case ... of ...` 構造の `when` 部分での暗黙的な等価比較でも許可されています。

条件付き選択

x と y が同じ型の 2 つの値で、`t` が `boolean` 型の値の場合、`if t then x else y` は、x および y と同じ型の式で、`t` が `TRUE` の場合に x、`t` が `FALSE` の場合に y を導出します。

I.5.2 ブールの演算

x が `boolean` の場合、`!x` はその論理否定です。

x と y が boolean の場合、`x && y` はそれらの AND の結果です。C 言語の場合と同様に、x が `FALSE` の場合、結果は `FALSE` であると y を計算することなく判定されます。

x と y が boolean の場合、`x || y` はそれらの OR の結果です。C 言語の場合と同様に、x が `TRUE` の場合、結果は `TRUE` であると y を計算することなく判定されます。

x と y が boolean の場合、`x ^ y` はそれらの排他的論理和の結果です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.5.3 Bitstring manipulation

The following bitstring manipulation functions are defined:

Bitstring length and most significant bit

If x is a bitstring, the bitstring length function Len(x) returns its length as an integer, and TopBit(x) is the leftmost bit of x (= x<Len(x)-1> using bitstring extraction).

Bitstring concatenation and replication

If x and y are bitstrings of lengths N and M respectively, then x:y is the bitstring of length N+M constructed by concatenating x and y in left-to-right order.

If x is a bitstring and n is an integer with n > 0, Replicate(x,n) is the bitstring of length n*Len(x) consisting of n copies of x concatenated together, and Zeros(n) = Replicate('0',n), Ones(n) = Replicate('1',n).

Bitstring extraction

The bitstring extraction operator extracts a bitstring from either another bitstring or an integer. Its syntax is x<integer_list>, where x is the integer or bitstring being extracted from, and <integer_list> is a list of integers enclosed in angle brackets rather than the usual parentheses. The length of the resulting bitstring is equal to the number of integers in <integer_list>. In x<integer_list>, each of the integers in <integer_list> must be:

- >= 0
- < Len(x) if x is a bitstring.

The definition of x<integer_list> depends on whether integer_list contains more than one integer. If it does, x<i,j,k,...,n> is defined to be the concatenation:

x<i> : x<j> : x<k> : ... : x<n>

If integer_list consists of just one integer i, x<i> is defined to be:

- if x is a bitstring, '0' if bit i of x is a zero and '1' if bit i of x is a one.
- if x is an integer, let y be the unique integer in the range 0 to 2^(i+1)-1 that is congruent to x modulo 2^(i+1). Then x<i> is '0' if y < 2^i and '1' if y >= 2^i.

Loosely, this second definition treats an integer as equivalent to a sufficiently long two's complement representation of it as a bitstring.

In <integer_list>, the notation i:j with i >= j is shorthand for the integers in order from i down to j, both ends inclusive. For example, instr<31:28> is shorthand for instr<31,30,29,28>.

The expression x<integer_list> is assignable provided x is an assignable bitstring and no integer appears more than once in <integer_list>. In particular, x<i> is assignable if x is an assignable bitstring and 0 <= i < Len(x).

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

I.5.3 ビットストリング操作

次のビットストリング操作関数が定義されています。

ビットストリング長および最上位ビット

x がビットストリングの場合、ビットストリング長関数 Len(x) はその長さを整数で返し、TopBit(x) はビットストリング抽出を使用して x の最も左のビット、つまり x<Len(x) - 1> を返します。

ビットストリングの連結と複製

x と y がそれぞれ長さ N および M のビットストリングの場合、x:y は、x と y を左から右への順序で連結させて算出した長さ N + M のビットストリングです。

x がビットストリングで、n が n > 0 の整数である場合、Replicate(x,n) はビットストリングで、その長さは n*Len(x) であり、連結された x の n 個のコピーから構成されます。また、Zeros(n) = Replicate('0',n)、Ones(n) = Replicate('1',n) です。

ビットストリング抽出

ビットストリング抽出演算子は、ビットストリングを別のビットストリングまたは整数から抽出します。構文は x<integer_list> です。x が抽出される整数またはビットストリングで、<integer_list> は通常の括弧の代わりに不等号で囲まれた整数のリストです。抽出されたビットストリングの長さは、<integer_list> に含まれている整数の数と同じです。x<integer_list> では、<integer_list> に含まれるそれぞれの整数は次の条件を満たす必要があります。

- >= 0
- x がビットストリングの場合、< Len(x)

x<integer_list> の定義は、integer_list に複数の整数が含まれるかどうかにより異なります。含まれる場合は、x<i,j,k,...,n> は次のような連結として定義されます。

x<i> : x<j> : x<k> : ... : x<n>

integer_list が 1 つの整数 i のみで構成される場合、x<i> は次のように定義されます。

- x がビットストリングの場合、x のビット x が 0 であれば '0'、x のビット i が 1 であれば '1'
- x が整数の場合、y は 0 ～ 2 ^ (i + 1) - 1 の範囲内で、x modulo 2 ^ (i + 1) と一致する一意の整数です。結果として x<i> は、y < 2 ^ i の場合は '0'、y >= 2 ^ i の場合は '1' です。

この 2 つ目の定義では、あいまいに整数を、その整数をビットストリングとして、十分な長さを持つ 2 の補数として表現したものと等価なものとして扱います。

<integer_list> で、i >= j の場合、表記 i:j は、i から j までの降順の整数（i および j 自身も含みます）の省略表記となります。例えば、instr<31:28> は instr<31,30,29,28> の省略表記です。

式 x<integer_list> は、x が代入可能なビットストリングで、同じ整数が <integer_list> に複数回現れない場合は、代入可能です。特に、x が代入可能なビットストリングで 0 <= i < Len(x) の場合、x<i> は代入可能です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Encoding diagrams for registers frequently show named bits or multi-bit fields. For example, the encoding diagram for the APSR shows its bit<31> as N. In such cases, the syntax APSR.N is used as a more readable synonym for APSR<31>.

Logical operations on bitstrings

If x is a bitstring, `NOT(x)` is the bitstring of the same length obtained by logically inverting every bit of x.

If x and y are bitstrings of the same length, `x AND y`, `x OR y`, and `x EOR y` are the bitstrings of that same length obtained by logically ANDing, ORing, and exclusive-ORing corresponding bits of x and y together.

Bitstring count

If x is a bitstring, `BitCount(x)` produces an integer result equal to the number of bits of x that are ones.

Testing a bitstring for being all zero or all ones

If x is a bitstring, `IsZero(x)` produces `TRUE` if all of the bits of x are zeros and `FALSE` if any of them are ones, and `IsZeroBit(x)` produces `'1'` if all of the bits of x are zeros and `'0'` if any of them are ones. `IsOnes(x)` and `IsOnesBit(x)` work in the corresponding way. So:

```
IsZero(x)    = (BitCount(x) == 0)
IsOnes(x)    = (BitCount(x) == Len(x))
IsZeroBit(x) = if IsZero(x) then '1' else '0'
IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

Lowest and highest set bits of a bitstring

If x is a bitstring, and `N = Len(x)`:

- `LowestSetBit(x)` is the minimum bit number of any of its bits that are ones. If all of its bits are zeros, `LowestSetBit(x) = N`.
- `HighestSetBit(x)` is the maximum bit number of any of its bits that are ones. If all of its bits are zeros, `HighestSetBit(x) = -1`.
- `CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x)` is the number of zero bits at the left end of x, in the range 0 to N.
- `CountLeadingSignBits(x) = CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>)` is the number of copies of the sign bit of x at the left end of x, excluding the sign bit itself, and is in the range 0 to N-1.

Zero-extension and sign-extension of bitstrings

If x is a bitstring and i is an integer, then `ZeroExtend(x,i)` is x extended to a length of i bits, by adding sufficient zero bits to its left. That is, if `i == Len(x)`, then `ZeroExtend(x,i) = x`, and if `i > Len(x)`, then:

```
ZeroExtend(x,i) = Replicate('0', i-Len(x)) : x
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

レジスタのエンコード図には、名前付きのビットや複数ビットフィールドが多く示されています。例えば、APSR のエンコード図は、ビット <31> を N として示しています。そのような場合、構文 APSR.N は APSR<31> のより読みやすい同義語として使用されます。

ビットストリングにおける論理演算

x がビットストリングの場合、`NOT(x)` は x のビットをすべて論理的に反転した同じ長さのビットストリングです。

x と y が同じ長さのビットストリングの場合、`x AND y`、`x OR y`、`x EOR y` は、x と y の対応するビットに論理積、論理和、排他的論理和を適用した、同じ長さのビットストリングです。

ビットストリングのカウント

x がビットストリングの場合、`BitCount(x)` は x に含まれている 1 のビットの数に等しい整数の結果を生成します。

ビットストリングがすべて 0 またはすべて 1 であるかどうかのテスト

x がビットストリングの場合、`IsZero(x)` は、x のすべてのビットが 0 のときに `TRUE` を、1 つでも 1 がある場合は `FALSE` を返します。`IsZeroBit(x)` は、x のビットがすべて 0 のときは `'1'` を、1 つでも 1 がある場合は `'0'` を生成します。`IsOnes(x)` と `IsOnesBit(x)` は、対応する方法で動作します。つまり、次のようになります。

```
IsZero(x)    = (BitCount(x) == 0)
IsOnes(x)    = (BitCount(x) == Len(x))
IsZeroBit(x) = if IsZero(x) then '1' else '0'
IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

ビットストリングでセットされている最下位および最上位ビット

x がビットストリングで `N = Len(x)` の場合、次のように関数が定義されます。

- `LowestSetBit(x)` は、含まれる 1 のビットの中で最小のビット番号です。すべてのビットが 0 の場合、`LowestSetBit(x) = N` です。
- `HighestSetBit(x)` は、含まれる 1 のビットの中で最大のビット番号です。すべてのビットが 0 の場合、`HighestSetBit(x) = - 1` です。
- `CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x)` では、範囲は 0 ～ N で、x の左端から数えた 0 のビットの数です。
- `CountLeadingSignBits(x) = CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>)` では、範囲は 0 ～ N − 1 で、x の左端から数えた、符号ビットと同じビットがコピーされている数（符号ビット自体を除く）です。

ビットストリングのゼロ拡張および符号拡張

x がビットストリングで i が整数の場合、`ZeroExtend(x,i)` は、必要な個数だけ 0 のビットを左側に加えて i ビットの長さに拡張された x です。つまり、`i == Len(x)` の場合、`ZeroExtend(x,i) = x` です。`i > Len(x)` の場合は次のように定義されます。

```
ZeroExtend(x,i) = Replicate('0', i-Len(x)) :x
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If x is a bitstring and i is an integer, then SignExtend(x,i) is x extended to a length of i bits, by adding sufficient copies of its leftmost bit to its left. That is, if i == Len(x), then SignExtend(x,i) = x, and if i > Len(x), then:

```
SignExtend(x,i) = Replicate(TopBit(x), i-Len(x)) : x
```

It is a pseudocode error to use either ZeroExtend(x,i) or SignExtend(x,i) in a context where it is possible that i < Len(x).

Converting bitstrings to integers

If x is a bitstring, SInt(x) is the integer whose two's complement representation is x:

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

UInt(x) is the integer whose unsigned representation is x:

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

Int(x, unsigned) returns either SInt(x) or UInt(x) depending on the value of its second argument:

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

I.5.4 Arithmetic

Most pseudocode arithmetic is performed on integer or real values, with operands being obtained by conversions from bitstrings and results converted back to bitstrings afterwards. As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

x がビットストリングで i が整数の場合、SignExtend(x,i) は、必要な個数だけ最も左のビットのコピーをその左側に加えて i ビットの長さに拡張された x です。つまり、i == Len(x) の場合、SignExtend(x,i) = x です。i > Len(x) の場合は次のように定義されます。

```
SignExtend(x,i) = Replicate(TopBit(x), i-Len(x)) :x
```

i < Len(x) の可能性がある場合に ZeroExtend(x,i) または SignExtend(x,i) を使用すると、擬似コードエラーとなります。

ビットストリングの整数への変換

x がビットストリングの場合、SInt(x) は 2 の補数表現が x となる整数です。

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

UInt(x) は、符号なし表現が x となる整数です。

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

Int(x, unsigned) は、2 つ目の引数の値に応じて、SInt(x) または UInt(x) を返します。

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

I.5.4 算術演算

ほとんどの擬似コード算術演算は整数値または実数値に対して実行され、オペランドはビットストリングからの変換により取得され、結果は再度ビットストリングに戻されます。これらのデータ型は制限のない数学的な型で、オーバフローや類似のエラーに関する問題は発生しません。

単項のプラス、マイナス、絶対値

x が整数または実数の場合、+x は変更されていない x、-x は符号が反転された x、Abs(x) は x の絶対値です。これらの 3 つはすべて、x と同じ型です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Unary plus, minus and absolute value

If x is an integer or real, then +x is x unchanged, -x is x with its sign reversed, and Abs(x) is the absolute value of x. All three are of the same type as x.

Addition and subtraction

If x and y are integers or reals, x+y and x-y are their sum and difference. Both are of type integer if x and y are both of type integer, and real otherwise.

Addition and subtraction are particularly common arithmetic operations in pseudocode, and so it is also convenient to have definitions of addition and subtraction acting directly on bitstring operands.

If x and y are bitstrings of the same length N = Len(x) = Len(y), then x+y and x-y are the least significant N bits of the results of converting them to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```
x+y = (SInt(x) + SInt(y))<N-1:0>
    = (UInt(x) + UInt(y))<N-1:0>
x-y = (SInt(x) - SInt(y))<N-1:0>
    = (UInt(x) - UInt(y))<N-1:0>
```

If x is a bitstring of length N and y is an integer, x+y and x-y are the bitstrings of length N defined by x+y = x + y<N-1:0> and x-y = x - y<N-1:0>. Similarly, if x is an integer and y is a bitstring of length M, x+y and x-y are the bitstrings of length M defined by x+y = x<M-1:0> + y and x-y = x<M-1:0> - y.

Comparisons

If x and y are integers or reals, then x == y, x != y, x < y, x <= y, x > y, and x >= y are equal, not equal, less than, less than or equal, greater than, and greater than or equal comparisons between them, producing boolean results. In the case of == and !=, this extends the generic definition applying to any two values of the same type to also act between integers and reals.

Multiplication

If x and y are integers or reals, then x * y is the product of x and y, of type integer if both x and y are of type integer and otherwise of type real.

Division and modulo

If x and y are integers or reals, then x / y is the result of dividing x by y, and is always of type real.

If x and y are integers, then x DIV y and x MOD y are defined by:

```
x DIV y = RoundDown(x / y)
x MOD y = x - y * (x DIV y)
```

It is a pseudocode error to use any x / y, x MOD y, or x DIV y in any context where y can be zero.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

加算と減算

x と y が整数または実数の場合、x+y と x-y はそれらの合計および差異です。x と y が両方とも integer 型の場合は、結果も両方とも integer 型です。それ以外の場合、結果は real 型です。

加算と減算は、擬似コードで特によく行われる算術演算です。このため、加算と減算の動作の定義をビットストリングの演算にも直接適用するのが便利です。

x と y が同じ長さのビットストリング、つまり N = Len(x) = Len(y) の場合、x+y と x-y は、それらを整数に変換して加算または減算を行った結果の下位 N ビットです。符号付きと符号なし変換は、いずれも同じ結果となります。

```
x+y = (SInt(x) + SInt(y))<N-1:0>
    = (UInt(x) + UInt(y))<N-1:0>
x-y = (SInt(x) - SInt(y))<N-1:0>
    = (UInt(x) - UInt(y))<N-1:0>
```

x が長さ N のビットストリングで、y が整数の場合、x+y と x-y は長さ N のビットストリングで、それぞれ x+y = x + y<N-1:0> および x-y = x - y<N-1:0> と定義されます。同様に、x が整数で y が長さ M のビットストリングの場合、x+y と x-y は長さ M のビットストリングで、それぞれ x+y = x<M-1:0>+ y および x-y = x<M-1:0> - y と定義されます。

比較

x と y が整数か実数である場合、x == y、x != y、x < y、x <= y?x > y、x >= y は、それらの間での等しい、等しくない、より小さい、以下、より大きい、以上の比較であり、ブールの結果を生成します。== および != の場合、同じ型の 2 つの値に適用される一般的な定義が拡張され、整数と実数との間にも適用されます。

操作

x と y が整数または実数の場合、x * y は x と y の積です。x と y の両方が integer 型の場合は結果も integer 型で、それ以外の場合、結果は real 型です。

除算とモジュロ

x と y が整数または実数の場合、x / y は x を y で除算した結果で、常に real 型です。

x と y が整数の場合、x DIV y と x MOD y は次のように定義されます。

```
x DIV y = RoundDown(x / y)
x MOD y = x - y * (x DIV y)
```

y が 0 である可能性がある場合に、x / y、x MOD y、または x DIV y を使用すると擬似コードエラーとなります。

平方根

x が整数または実数の場合、Sqrt(x) は平方根で、常に実数型です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Square Root

If x is an integer or a real, Sqrt(x) is its square root, and is always of type real.

Rounding and aligning

If x is a real:

- RoundDown(x) produces the largest integer n such that n <= x.
- RoundUp(x) produces the smallest integer n such that n >= x.
- RoundTowardsZero(x) produces RoundDown(x) if x > 0.0, 0 if x == 0.0, and RoundUp(x) if x < 0.0.

If x and y are integers, Align(x,y) = y * (x DIV y) is an integer.

If x is a bitstring and y is an integer, Align(x,y) = (Align(UInt(x),y))<Len(x)-1:0> is a bitstring of the same length as x.

It is a pseudocode error to use either form of Align(x,y) in any context where y can be 0. In practice, Align(x,y) is only used with y a constant power of two, and the bitstring form used with y = 2^n has the effect of producing its argument with its n low-order bits forced to zero.

Scaling

If n is an integer, 2^n is the result of raising 2 to the power n and is of type real.

If x and n are integers, then:

- x << n = RoundDown(x * 2^n)
- x >> n = RoundDown(x * 2^(-n)).

Maximum and minimum

If x and y are integers or reals, then Max(x,y) and Min(x,y) are their maximum and minimum respectively. Both are of type integer if both x and y are of type integer and of type real otherwise.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

丸めとアライン

x が実数の場合、次のように定義されます。

- RoundDown(x) は、n <= x である最大の整数 n を生成します。
- RoundUp(x) は、n >= x である最小の整数 n を生成します。
- RoundTowardsZero(x) は、x > 0.0 の場合は RoundDown(x)、x == 0.0 の場合は 0、x < 0.0 の場合は RoundUp(x) を生成します。

x と y が整数の場合、Align(x,y) = y * (x DIV y) は整数です。

x がビットストリングで y が整数の場合、Align(x,y) = (Align(UInt(x),y))<Len(x)-1:0> は x と同じ長さのビットストリングです。

y が 0 である可能性がある場合、いずれかの形式の Align(x,y) を使用すると擬似コードエラーになります。実際には、Align(x,y) は y が 2 の定数乗である場合にのみ使用され、y = 2^n とともに使用されるビットストリング形式には、下位ビット n を強制的に 0 にして引数を生成する効果があります。

スケーリング

n が整数の場合、2^n は 2 を n 乗した結果になり、real 型です。

x と n が整数の場合、次のように定義されます。

- x << n = RoundDown(x * 2^n)
- x >> n = RoundDown(x * 2^(-n))

最大値と最小値

x と y が整数または実数の場合、Max(x,y) と Min(x,y) は最大値および最小値です。x と y が両方とも integer 型の場合は、結果も両方とも integer 型です。それ以外の場合、結果は real 型です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.6 Statements and program structure

This section describes the control statements used in the pseudocode.

I.6.1 Simple statements

The following simple statements must all be terminated with a semicolon, as shown.

Assignments

An assignment statement takes the form:

```
<assignable_expression> = <expression>;
```

Procedure calls

A procedure call takes the form:

```
<procedure_name>(<arguments>);
```

Return statements

A procedure return takes the form:

```
return;
```

and a function return takes the form:

```
return <expression>;
```

where `<expression>` is of the type the function prototype line declared.

UNDEFINED

The statement:

```
UNDEFINED;
```

indicates a special case that replaces the behavior defined by the current pseudocode (apart from behavior required to determine that the special case applies). The replacement behavior is that the Undefined Instruction exception is taken.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.6 文とプログラム構造

このセクションでは、擬似コードで使用される制御文について説明します。

I.6.1 単純な文

以下に示す単純な文は、すべてセミコロンで終わる必要があります。

代入

代入文は次の形式です。

```
<assignable_expression> = <expression>;
```

プロシージャ呼び出し

プロシージャ呼び出しは次の形式です。

```
<procedure_name>(<arguments>);
```

リターン文

プロシージャからの復帰は次の形式です。

```
return;
```

関数の戻り値は次の形式です。

```
return <expression>;
```

ここで、`<expression>` は関数プロトタイプ行で宣言された型です。

未定義

この文は次の形式です。

```
UNDEFINED;
```

現在の擬似コードにより定義された動作を置き換える特殊なケースのひとつであることを示します（特殊なケースが適用されることを判定するために必要な動作を除きます）。置き換え動作では、未定義命令例外が取得されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

UNPREDICTABLE

The statement:

```
UNPREDICTABLE;
```

indicates a special case that replaces the behavior defined by the current pseudocode (apart from behavior required to determine that the special case applies). The replacement behavior is not architecturally defined and must not be relied upon by software. It must not constitute a security hole or halt or hang the system, and must not be promoted as providing any useful information to software.

SEE...

The statement:

```
SEE <reference>;
```

indicates a special case that replaces the behavior defined by the current pseudocode (apart from behavior required to determine that the special case applies). The replacement behavior is that nothing occurs as a result of the current pseudocode because some other piece of pseudocode defines the required behavior. The `<reference>` indicates where that other pseudocode can be found.

It usually refers to another instruction, but can also refer to another encoding or note of the same instruction.

IMPLEMENTATION_DEFINED

The statement:

```
IMPLEMENTATION_DEFINED <text>;
```

indicates a special case that specifies that the behavior is IMPLEMENTATION DEFINED. Following text can give more information.

SUBARCHITECTURE_DEFINED

The statement:

```
SUBARCHITECTURE_DEFINED <text>;
```

indicates a special case that specifies that the behavior is SUBARCHITECTURE DEFINED. Following text can give more information.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

予測不能

この文は次の形式です。

```
UNPREDICTABLE;
```

現在の擬似コードにより定義された動作を置き換える特殊なケースのひとつであることを示します（特殊なケースが適用されることを判定するために必要な動作を除きます）。置き換え動作は、アーキテクチャ上では定義されていません。ソフトウェアは、この動作に依存しないようにする必要があります。置き換え動作が、セキュリティホールや、システムの停止またはハングの原因にならないようにする必要があります。また、この動作がソフトウェアに有用な情報を提供するものとは示さないことが必要です。

SEE...

この文は次の形式です。

```
SEE <reference>;
```

現在の擬似コードにより定義された動作を置き換える特殊なケースのひとつであることを示します（特殊なケースが適用されることを判定するために必要な動作を除きます）。置き換え動作では、現在の擬似コードの結果としては何も起きません。必要な動作は別の擬似コードで定義されています。`<reference>` は、擬似コードのその部分がどこにあるのかを示しています。

通常この場合は別の命令を参照しますが、同じ命令の別のエンコードまたは注記を参照することもできます。

IMPLEMENTATION_DEFINED

この文は次の形式です。

```
IMPLEMENTATION_DEFINED <text>;
```

動作が実装定義となる特殊なケースのひとつであることを示します。詳細は、文の後に続くテキストに記載されています。

SUBARCHITECTURE_DEFINED

この文は次の形式です。

```
SUBARCHITECTURE_DEFINED <text>;
```

動作がサブアーキテクチャ定義となる特殊なケースのひとつであることを示します。詳細は、文の後に続くテキストに記載されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.6.2 Compound statements

Indentation is normally used to indicate structure in compound statements. The statements contained in structures such as `if ... then ... else ...` or procedure and function definitions are indented more deeply than the statement itself, and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level.

if ... then ... else ...

A multi-line `if ... then ... else ...` structure takes the form:

```
if <boolean_expression> then
    <statement 1>
    <statement 2>
    ...
    <statement n>
elsif <boolean_expression> then
    <statement a>
    <statement b>
    ...
    <statement z>
else
    <statement A>
    <statement B>
    ...
    <statement Z>
```

The block of lines consisting of `elsif` and its indented statements is optional, and multiple such blocks can be used.

The block of lines consisting of `else` and its indented statements is optional.

Abbreviated one-line forms can be used when there are only simple statements in the `then` part and (if present) the `else` part, such as:

```
if <boolean_expression> then <statement 1>
if <boolean_expression> then <statement 1> else <statement A>
if <boolean_expression> then <statement 1> <statement 2> else <statement A>
```

Note

In these forms, `<statement 1>`, `<statement 2>` and `<statement A>` must be terminated by semicolons. This and the fact that the `else` part is optional are differences from the `if ... then ... else ...` expression.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.6.2 複合文

インデントは通常、複合文の構造を示すために使用されます。`if ... then ... else ...` などの構造に含まれる文、またはプロシージャや関数の定義は、文自体よりもより深くインデントされており、それらの終わりは元のインデントレベルまたはそれ以下のレベルに戻ることで示されます。

インデントは通常、レベルごとに 4 つのスペースにより示されます。

if ... then ... else ...

複数行の `if ... then ... else ...` 構造は、次の形式です。

```
if <boolean_expression> then
    <statement 1>
    <statement 2>
    ...
    <statement n>
elsif <boolean_expression> then
    <statement a>
    <statement b>
    ...
    <statement z>
else
    <statement A>
    <statement B>
    ...
    <statement Z>
```

`elsif` と、その後のインデントされた文で構成されるブロックはオプションです。また、そのようなブロックを複数使用することができます。

`else` と、その後のインデントされた文で構成されるブロックはオプションです。

`then` および `else`（存在する場合）部分に単純な文しか存在しない場合は、次のように省略された 1 行の形式を使用することができます。

```
if <boolean_expression> then <statement 1>
if <boolean_expression> then <statement 1> else <statement A>
if <boolean_expression> then <statement 1> <statement 2> else <statement A>
```

注

これらの形式では、`<statement 1>`、`<statement 2>`、および `<statement A>` は、セミコロンで終了する必要があります。この点と、`else` 部分がオプションであることが、`if ... then ... else ...` 式との相違点です。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

repeat ... until ...

A `repeat ... until ...` structure takes the form:

```
repeat
    <statement 1>
    <statement 2>
    ...
    <statement n>
until <boolean_expression>;
```

while ... do

A `while ... do` structure takes the form:

```
while <boolean_expression>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

for ...

A `for ...` structure takes the form:

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

case ... of ...

A `case ... of ...` structure takes the form:

```
case <expression> of
    when <constant values>
        <statement 1>
        <statement 2>
        ...
        <statement n>
    ... more "when" groups ...
    otherwise
        <statement A>
        <statement B>
        ...
        <statement Z>
```

where `<constant values>` consists of one or more constant values of the same type as `<expression>`, separated by commas. Abbreviated one line forms of `when` and `otherwise` parts can be used when they contain only simple statements.

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

repeat ... until ...

`repeat ... until ...` 構造は、次の形式です。

```
repeat
    <statement 1>
    <statement 2>
    ...
    <statement n>
until <boolean_expression>;
```

while ... do

`while ... do` 構造は、次の形式です。

```
while <boolean_expression>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

for ...

`for ...` 構造は、次の形式です。

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

case ... of ...

`case ... of ...` 構造は、次の形式です。

```
case <expression> of
    when <constant values>
        <statement 1>
        <statement 2>
        ...
        <statement n>
    ... more "when" groups ...
    otherwise
        <statement A>
        <statement B>
        ...
        <statement Z>
```

ここで、`<constant values>` は、`<expression>` と同じ型の 1 つ以上の定数値を、カンマで区切って並べたものです。`when` と `otherwise` 部分が単純な文のみを含む場合は、それらの部分が短縮された 1 行の形式を使用することができます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

If <expression> has a bitstring type, <constant values> can also include bitstring constants containing 'x' bits. For details see *Equality and non-equality testing* on page AppxI-11.

Procedure and function definitions

A procedure definition takes the form:

```
<procedure name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

where the <argument prototypes> consists of zero or more argument definitions, separated by commas. Each argument definition consists of a type name followed by the name of the argument.

Note

This first prototype line is not terminated by a semicolon. This helps to distinguish it from a procedure call.

A function definition is similar, but also declares the return type of the function:

```
<return type> <function name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

An array-like function is similar, but with square brackets:

```
<return type> <function name>[<argument prototypes>]
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

An array-like function also usually has an assignment prototype:

```
<function name>[<argument prototypes>] = <value prototypes>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

I.6.3 Comments

Two styles of pseudocode comment exist:

- // starts a comment that is terminated by the end of the line.
- /* starts a comment that is terminated by */.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

<expression> がビットストリング型の場合、<constant values> には 'x' ビットを含むビットストリング定数も含めることができます。詳細については、*等価と非等価のテスト* P. 付録 I-11 を参照して下さい。

プロシージャと関数の定義

プロシージャの定義は次の形式です。

```
<procedure name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

ここで、<argument prototypes> はカンマで区切られた 0 個以上の引数の定義です。それぞれの引数の定義は、型名と、それに続く引数名で構成されます。

注

最初のプロトタイプ行はセミコロンで終わりません。これにより、プロシージャ呼び出しと区別することができます。

関数の定義も同様ですが、関数の戻り値の型も宣言されます。

```
<return type> <function name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

配列様の関数も同様の形式ですが、この場合は角括弧が使用されます。

```
<return type> <function name>[<argument prototypes>]
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

また、配列様の関数には、一般に代入プロトタイプも含まれます。

```
<function name>[<argument prototypes>] = <value prototypes>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

I.6.3 コメント

擬似コードのコメントには 2 つの形式があります。

- // で始まるコメントは、行の終わりで終了します。
- /* で始まるコメントは、*/ で終了します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.7 Miscellaneous helper procedures and functions

The functions described in this section are not part of the pseudocode specification. They are miscellaneous *helper* procedures and functions used by pseudocode that are not described elsewhere in this manual. Each has a brief description and a pseudocode prototype, except that the prototype is omitted where it is identical to the section title.

I.7.1 ArchVersion()

This function returns the major version number of the architecture.

```
integer ArchVersion()
```

I.7.2 BadReg()

This function performs the check for the register numbers 13 and 15 that are not permitted for many Thumb register specifiers.

```
// BadReg()
// ========

boolean BadReg(integer n)
    return n == 13 || n == 15;
```

I.7.3 Breakpoint()

This procedure causes a debug breakpoint to occur.

I.7.4 CallSupervisor()

This procedure causes the appropriate exception to occur to call a privileged supervisor.

In all architecture variants and profiles described in this manual, `CallSupervisor()` causes a Supervisor Call exception.

I.7.5 Coproc_Accepted()

This function determines, for a coprocessor and one of its coprocessor instructions:

- Whether access to the coprocessor is permitted by the CPACR and, if the Security Extensions are implemented, the NSACR.
- If access is permitted, whether the instruction is accepted by the coprocessor. The coprocessor architecture definition specifies which instructions it accepts and in what circumstances.

It returns TRUE if access is permitted and the coprocessor accepts the instruction, and FALSE otherwise.

```
boolean Coproc_Accepted(integer cp_num, bits(32) instr)
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

I.7 その他のヘルパプロシージャおよび関数

このセクションで説明されている関数は擬似コード仕様の一部ではありません。これらは、本書の他の場所で説明されていない擬似コードによって使用される、その他のヘルパプロシージャおよび関数です。それぞれには簡単な説明と擬似コードのプロトタイプが示されています。ただし、セクションタイトルと同一の場合にはプロトタイプは省略されています。

I.7.1 ArchVersion()

この関数は、アーキテクチャのメジャーバージョン番号を返します。

```
integer ArchVersion()
```

I.7.2 BadReg()

この関数は、多くの Thumb レジスタ指定子で許可されていないレジスタ番号 13 と 15 に対するチェックを行います。

```
// BadReg()
// ========

boolean BadReg(integer n)
    return n == 13 || n == 15;
```

I.7.3 Breakpoint()

このプロシージャは、デバッグブレークポイントを引き起こします。

I.7.4 CallSupervisor()

このプロシージャは、特権スーパバイザを呼び出すために適切な例外を発生させます。

本書で説明されているすべてのアーキテクチャバリアントとプロファイルにおいて、`CallSupervisor()` によりスーパバイザコール例外が発生します。

I.7.5 Coproc_Accepted()

この関数は、コプロセッサとその命令のいずれかについて次の項目を決定します。

- コプロセッサへのアクセスが、CPACR および NSACR（セキュリティ拡張機能が実装されている場合）によって許可されるかどうか
- アクセスが許可される場合、命令がコプロセッサによって受け入れられるかどうか。コプロセッサのアーキテクチャ定義では、受け入れられる命令およびその条件を指定します。

アクセスが許可されてコプロセッサが命令を受け入れた場合、この関数は TRUE を返します。それ以外の場合は FALSE を返します。

```
boolean Coproc_Accepted(integer cp_num, bits(32) instr)
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.7.6 Coproc_DoneLoading()

This function determines for an LDC instruction whether enough words have been loaded.

```
boolean Coproc_DoneLoading(integer cp_num, bits(32) instr)
```

I.7.7 Coproc_DoneStoring()

This function determines for an STC instruction whether enough words have been stored.

```
boolean Coproc_DoneStoring(integer cp_num, bits(32) instr)
```

I.7.8 Coproc_GetOneWord()

This function obtains the word for an MRC instruction from the coprocessor.

```
bits(32) Coproc_GetOneWord(integer cp_num, bits(32) instr)
```

I.7.9 Coproc_GetTwoWords()

This function obtains the two words for an MRRC instruction from the coprocessor.

```
(bits(32), bits(32)) Coproc_GetTwoWords(integer cp_num, bits(32) instr)
```

I.7.10 Coproc_GetWordToStore()

This function obtains the next word to store for an STC instruction from the coprocessor

```
bits(32) Coproc_GetWordToStore(integer cp_num, bits(32) instr)
```

I.7.11 Coproc_InternalOperation()

This procedure instructs a coprocessor to perform the internal operation requested by a CDP instruction.

```
Coproc_InternalOperation(integer cp_num, bits(32) instr)
```

I.7.12 Coproc_SendLoadedWord()

This procedure sends a loaded word for an LDC instruction to the coprocessor.

```
Coproc_SendLoadedWord(bits(32) word, integer cp_num, bits(32) instr)
```

I.7.13 Coproc_SendOneWord()

This procedure sends the word for an MCR instruction to the coprocessor.

```
Coproc_SendOneWord(bits(32) word, integer cp_num, bits(32) instr)
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.7.6 Coproc_DoneLoading()

この関数は、LDC 命令で十分な数のワードがロードされたかどうかを判定します。

```
boolean Coproc_DoneLoading(integer cp_num, bits(32) instr)
```

I.7.7 Coproc_DoneStoring()

この関数は、STC 命令で十分な数のワードがストアされたかどうかを判定します。

```
boolean Coproc_DoneStoring(integer cp_num, bits(32) instr)
```

I.7.8 Coproc_GetOneWord()

この関数は、MRC 命令のワードをコプロセッサから取得します。

```
bits(32) Coproc_GetOneWord(integer cp_num, bits(32) instr)
```

I.7.9 Coproc_GetTwoWords()

この関数は、MRRC 命令の 2 ワードをコプロセッサから取得します。

```
(bits(32), bits(32)) Coproc_GetTwoWords(integer cp_num, bits(32) instr)
```

I.7.10 Coproc_GetWordToStore()

この関数は、STC 命令でストアする次のワードをコプロセッサから取得します。

```
bits(32) Coproc_GetWordToStore(integer cp_num, bits(32) instr)
```

I.7.11 Coproc_InternalOperation()

このプロシージャは、CDP 命令により要求された内部操作を行うようにコプロセッサに指示します。

```
Coproc_InternalOperation(integer cp_num, bits(32) instr)
```

I.7.12 Coproc_SendLoadedWord()

このプロシージャは、LDC 命令のためにロードされたワードをコプロセッサに送ります。

```
Coproc_SendLoadedWord(bits(32) word, integer cp_num, bits(32) instr)
```

I.7.13 Coproc_SendOneWord()

このプロシージャは、MCR 命令のワードをコプロセッサに送ります。

```
Coproc_SendOneWord(bits(32) word, integer cp_num, bits(32) instr)
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.7.14 Coproc_SendTwoWords()

This procedure sends the two words for an MCRR instruction to the coprocessor.

```
Coproc_SendTwoWords(bits(32) word1, bits(32) word2, integer cp_num, bits(32) instr)
```

I.7.15 EndOfInstruction()

This procedure terminates processing of the current instruction.

I.7.16 GenerateAlignmentException()

This procedure generates the appropriate exception for an alignment error.

In all architecture variants and profiles described in this manual, GenerateAlignmentException() generates a Data Abort exception.

I.7.17 GenerateCoprocessorException()

This procedure generates the appropriate exception for a rejected coprocessor instruction.

In all architecture variants and profiles described in this manual, GenerateCoprocessorException() generates an Undefined Instruction exception.

I.7.18 GenerateIntegerZeroDivide()

This procedure generates the appropriate exception for a division by zero in the integer division instructions SDIV and UDIV.

In the ARMv7-R profile, GenerateIntegerZeroDivide() generates an Undefined Instruction exception. The integer division instructions do not exist in any other architecture variant or profile described in this manual, so the GenerateIntegerZeroDivide() procedure is never called in those variants and profiles.

I.7.19 HaveMPExt()

This procedure returns true if the MP Extensions are implemented.

```
boolean HaveMPExt()
```

I.7.20 Hint_Debug()

This procedure supplies a hint to the debug system.

```
Hint_Debug(bits(4) option)
```


Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

I.7.14 Coproc_SendTwoWords()

このプロシージャは、MCRR 命令の 2 ワードをコプロセッサに送ります。

```
Coproc_SendTwoWords(bits(32) word1, bits(32) word2, integer cp_num, bits(32) instr)
```

I.7.15 EndOfInstruction()

このプロシージャは、現在の命令の処理を終了します。

I.7.16 GenerateAlignmentException()

このプロシージャは、アライメントエラーに対して該当する例外を生成します。

本書で説明されているすべてのアーキテクチャバリアントとプロファイルにおいて、GenerateAlignmentException() によりデータアボート例外が発生します。

I.7.17 GenerateCoprocessorException()

このプロシージャは、拒否されたコプロセッサ命令に対して該当する例外を生成します。

本書で説明されているすべてのアーキテクチャバリアントとプロファイルにおいて、GenerateCoprocessorException() により未定義命令例外が発生します。

I.7.18 GenerateIntegerZeroDivide()

このプロシージャは、整数除算命令 SDIV および UDIV での 0 による除算に対して、該当する例外を生成します。

ARMv7-R プロファイルでは、GenerateIntegerZeroDivide() により未定義命令例外が発生します。本書で説明されている他のアーキテクチャバリアントまたはプロファイルには、整数除算命令は存在しません。このため、それらのバリアントとプロファイルで GenerateIntegerZeroDivide() プロシージャが呼び出されることはありません。

I.7.19 HaveMPExt()

このプロシージャは、MP 拡張機能が実装されている場合に TRUE を返します。

```
boolean HaveMPExt()
```

I.7.20 Hint_Debug()

このプロシージャは、デバッグシステムにヒントを提供します。

```
Hint_Debug(bits(4) option)
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.7.21 Hint_PreloadData()

This procedure performs a *preload data* hint.

```
Hint_PreloadData(bits(32) address)
```

I.7.22 Hint_PreloadDataForWrite()

This procedure performs a *preload data* hint with a probability that the use will be for a write.

```
Hint_PreloadDataForWrite(bits(32) address
```

I.7.23 Hint_PreloadInstr()

This procedure performs a *preload instructions* hint.

```
Hint_PreloadInstr(bits(32) address)
```

I.7.24 Hint_Yield()

This procedure performs a *Yield* hint.

I.7.25 IntegerZeroDivideTrappingEnabled()

This function returns TRUE if the trapping of divisions by zero in the integer division instructions `SDIV` and `UDIV` is enabled, and FALSE otherwise.

In the ARMv7-R profile, this is controlled by the SCTLR.DZ bit, see *c1, System Control Register (SCTLR)* on page B4-45. TRUE is returned if the bit is 1 and FALSE if it is 0. This function is never called in the A profile.

```
boolean IntegerZeroDivideTrappingEnabled()
```

I.7.26 IsExternalAbort()

This function returns TRUE if the abort currently being processed is an external abort and FALSE otherwise. It is only used in abort exception entry pseudocode.

```
boolean IsExternalAbort()
```

I.7.27 JazelleAcceptsExecution()

This function indicates whether Jazelle hardware will take over execution when a `BXJ` instruction is executed.

```
boolean JazelleAcceptsExecution()
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.7.21 Hint_PreloadData()

このプロシージャは、*データのプリロード*のヒントを実行します。

```
Hint_PreloadData(bits(32) address)
```

I.7.22 Hint_PreloadDataForWrite()

このプロシージャは、書き込み用に使用される可能性のある*データのプリロード*のヒントを実行します。

```
Hint_PreloadDataForWrite(bits(32) address
```

I.7.23 Hint_PreloadInstr()

このプロシージャは、*命令のプリロード*のヒントを実行します。

```
Hint_PreloadInstr(bits(32) address)
```

I.7.24 Hint_Yield()

このプロシージャは*イールド*のヒントを実行します。

I.7.25 IntegerZeroDivideTrappingEnabled()

整数除算命令 `SDIV` および `UDIV` で 0 による除算のトラップが許可されている場合、この関数は TRUE を返します。それ以外の場合は FALSE を返します。

ARMv7-R プロファイルでは、これは SCTLR.DZ ビットによって制御されます。*c1、システム制御レジスタ (SCTLR)* P. B4-45 を参照して下さい。このビットが 1 の場合は TRUE が返され、0 の場合は FALSE が返されます。A プロファイルでこの関数が呼び出されることはありません。

```
boolean IntegerZeroDivideTrappingEnabled()
```

I.7.26 IsExternalAbort()

現在処理されているアボートが外部アボートである場合、この関数は TRUE を返します。それ以外の場合は FALSE を返します。この関数は、アボート例外エントリ擬似コードでのみ使用されます。

```
boolean IsExternalAbort()
```

I.7.27 JazelleAcceptsExecution()

この関数は、`BXJ` 命令が実行される際に、Jazelle ハードウェアが実行を引き継ぐかどうかを示します。

```
boolean JazelleAcceptsExecution()
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

I.7.28 MemorySystemArchitecture()

This function returns a value indicating which memory system architecture is in use on the system.

```
enumeration MemArch {MemArch_VMSA, MemArch_PMSA};

MemArch MemorySystemArchitecture()
```

I.7.29 ProcessorID()

This function returns an integer that uniquely identifies the executing processor in the system.

```
integer ProcessorID()
```

I.7.30 RemapRegsHaveResetValues()

This function returns TRUE if the remap registers PRRR and NMRR have their IMPLEMENTATION DEFINED reset values, and FALSE otherwise.

```
boolean RemapRegsHaveResetValues()
```

I.7.31 SwitchToJazelleExecution()

This procedure passes control of execution to Jazelle hardware (for a `BXJ` instruction).

I.7.32 ThisInstr()

This function returns the currently-executing instruction. It is only used on 32-bit instruction encodings at present.

```
bits(32) ThisInstr()
```

I.7.33 UnalignedSupport()

This function returns TRUE if the processor currently provides support for unaligned memory accesses, or FALSE otherwise. This is always TRUE in ARMv7, controllable by the SCTLR.U bit in ARMv6, and always FALSE before ARMv6.

```
boolean UnalignedSupport()
```

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

I.7.28 MemorySystemArchitecture()

この関数は、システム上で使用されているメモリシステム アーキテクチャを示す値を返します。

```
enumeration MemArch {MemArch_VMSA, MemArch_PMSA};

MemArch MemorySystemArchitecture()
```

I.7.29 ProcessorID()

この関数は、システムで実行中のプロセッサを一意に特定する整数を返します。

```
integer ProcessorID()
```

I.7.30 RemapRegsHaveResetValues()

再マップレジスタ PRRR および NMRR が実装定義のリセット時の値を持つ場合、この関数は TRUE を返します。それ以外の場合は FALSE を返します。

```
boolean RemapRegsHaveResetValues()
```

I.7.31 SwitchToJazelleExecution()

このプロシージャは、実行の制御を Jazelle ハードウェアに受け渡します（`BXJ` 命令の場合）。

I.7.32 ThisInstr()

この関数は、現在実行されている命令を返します。現在のところ、これは 32 ビット命令エンコードでのみ使用されます。

```
bits(32) ThisInstr()
```

I.7.33 UnalignedSupport()

プロセッサが現在アンアライインド メモリアクセスをサポートしている場合、この関数は TRUE を返します。それ以外の場合は FALSE を返します。ARMv7 では常に TRUE で、ARMv6 では SCTLR.U ビットにより制御可能です。ARMv6 以前では常に FALSE です。

```
boolean UnalignedSupport()
```

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Appendix J
Pseudocode Index

This appendix provides an index to pseudocode operators and functions that occur elsewhere in this manual. It contains the following sections:

- *Pseudocode operators and keywords* on page AppxJ-2
- *Pseudocode functions and procedures* on page AppxJ-6.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

付録 J
擬似コード索引

この付録は、本書の他の場所で使用されている擬似コードの演算子と関数の索引です。本章は次のセクションから構成されています。

- *擬似コードの演算子とキーワード* P. 付録 J-2
- *擬似コードの関数とプロシージャ* P. 付録 J-6

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

J.1 Pseudocode operators and keywords

Table J-1 lists the pseudocode operators and keywords, and is an index to their descriptions:

Table J-1 Pseudocode operators and keywords

Operator	Meaning	See
-	Unary minus on integers or reals	*Unary plus, minus and absolute value* on page AppxI-15
-	Subtraction of integers, reals and bitstrings	*Addition and subtraction* on page AppxI-15
+	Unary plus on integers or reals	*Unary plus, minus and absolute value* on page AppxI-15
+	Addition of integers, reals and bitstrings	*Addition and subtraction* on page AppxI-15
(...)	Around arguments of procedure	*Procedure calls* on page AppxI-17, *Procedure and function definitions* on page AppxI-21
(...)	Around arguments of function	*General expression syntax* on page AppxI-9, *Procedure and function definitions* on page AppxI-21
.	Extract named member from a list	*Lists* on page AppxI-7
.	Extract named bit or field from a register	*Bitstring extraction* on page AppxI-12
!	Boolean NOT	*Operations on booleans* on page AppxI-11
!=	Compare for non-equality (any type)	*Equality and non-equality testing* on page AppxI-11
!=	Compare for non-equality (between integers and reals)	*Comparisons* on page AppxI-15
&&	Boolean AND	*Operations on booleans* on page AppxI-11
*	Multiplication of integers and reals	*Multiplication* on page AppxI-15
/	Division of integers and reals (real result)	*Division and modulo* on page AppxI-15
/*...*/	Comment delimiters	*Comments* on page AppxI-21
//	Introduces comment terminated by end of line	*Comments* on page AppxI-21

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

J.1 擬似コードの演算子とキーワード

表 J-1 は、擬似コードの演算子とキーワードを一覧表記し、それぞれの説明のあるページを示しています。

表 J-1 擬似コードの演算子とキーワード

演算子	意味	参照先
-	整数または実数に対する単項のマイナス	*単項のプラス、マイナス、絶対値* P. 付録 I-14
-	整数、実数、ビットストリングの減算	*加算と減算* P. 付録 I-15
+	整数または実数に対する単項のプラス	*単項のプラス、マイナス、絶対値* P. 付録 I-14
+	整数、実数、ビットストリングの加算	*加算と減算* P. 付録 I-15
(...)	プロシージャの引数を囲む	*プロシージャ呼び出し* P. 付録 I-17, *プロシージャと関数の定義* P. 付録 I-21
(...)	関数の引数を囲む	*一般的な式の構文* P. 付録 I-9, *プロシージャと関数の定義* P. 付録 I-21
.	リストから名前付きのメンバを抽出する	*リスト* P. 付録 I-7
.	レジスタから名前付きのビットやフィールドを抽出する	*ビットストリング抽出* P. 付録 I-12
!	ブールの論理否定	*ブールの演算* P. 付録 I-11
!=	等しくないかどうかの比較（あらゆる型）	*等価と非等価のテスト* P. 付録 I-11
!=	等しくないどうかの比較（整数と実数）	*比較* P. 付録 I-15
&&	ブールの論理積	*ブールの演算* P. 付録 I-11
*	整数と実数の乗算	*操作* P. 付録 I-15
/	整数と実数の除算（実数の結果）	*除算とモジュロ* P. 付録 I-15
/*...*/	コメントの区切り文字	*コメント* P. 付録 I-21
//	行の終わりで終了するコメントの開始	*コメント* P. 付録 I-21

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-1 Pseudocode operators and keywords (continued)

Operator	Meaning	See
:	Bitstring concatenation	*Bitstring concatenation and replication* on page AppxI-12
:	Integer range in bitstring extraction operator	*Bitstring extraction* on page AppxI-12
[...]	Around array index	*Arrays* on page AppxI-8
[...]	Around arguments of array-like function	*General expression syntax* on page AppxI-9, *Procedure and function definitions* on page AppxI-21
^	Boolean exclusive-OR	*Operations on booleans* on page AppxI-11
||	Boolean OR	*Operations on booleans* on page AppxI-11
<	*Less than* comparison of integers and reals	*Comparisons* on page AppxI-15
<...>	Extraction of specified bits of bitstring or integer	*Bitstring extraction* on page AppxI-12
<<	Multiply integer by power of 2 (with rounding towards -infinity)	*Scaling* on page AppxI-16
<=	*Less than or equal* comparison of integers and reals	*Comparisons* on page AppxI-15
=	Assignment	*Assignments* on page AppxI-17
==	Compare for equality (any type)	*Equality and non-equality testing* on page AppxI-11
==	Compare for equality (between integers and reals)	*Comparisons* on page AppxI-15
>	*Greater than* comparison of integers and reals	*Comparisons* on page AppxI-15
>=	*Greater than or equal* comparison of integers and reals	*Comparisons* on page AppxI-15
>>	Divide integer by power of 2 (with rounding towards -infinity)	*Scaling* on page AppxI-16
2^N	Power of two (real result)	*Scaling* on page AppxI-16

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 J-1 擬似コードの演算子とキーワード （続き）

演算子	意味	参照先
:	ビットストリングの連結	*ビットストリングの連結と複製* P. 付録 I-12
:	ビットストリング抽出演算子における整数範囲	*ビットストリング抽出* P. 付録 I-12
[...]	配列のインデクスを囲む	*配列* P. 付録 I-8
[...]	配列様関数の引数を囲む	*一般的な式の構文* P. 付録 I-9, *プロシージャと関数の定義* P. 付録 I-21
^	ブールの排他的論理和	*ブールの演算* P. 付録 I-11
||	ブールの論理和	*ブールの演算* P. 付録 I-11
<	整数と実数で*左辺が右辺より小さいかどうか*の比較	*比較* P. 付録 I-15
<...>	ビットストリングまたは整数の指定したビットを抽出する	*ビットストリング抽出* P. 付録 I-12
<<	整数に 2 のべき乗を乗算する（- 無限大への丸め）	*スケーリング* P. 付録 I-16
<=	整数と実数で*左辺が右辺より小さいまたは等しいかどうか*の比較	*比較* P. 付録 I-15
=	代入	*代入* P. 付録 I-17
==	等しいかどうかの比較（あらゆる型）	*等価と非等価のテスト* P. 付録 I-11
==	等しいかどうかの比較（整数と実数）	*比較* P. 付録 I-15
>	整数と実数で*左辺が右辺より大きいかどうか*の比較	*比較* P. 付録 I-15
>=	整数と実数で*左辺が右辺より大きいまたは等しいかどうか*の比較	*比較* P. 付録 I-15
>>	整数を 2 のべき乗で除算（- 無限大への丸め）	*スケーリング* P. 付録 I-16
2^N	2 のべき乗（実数の結果）	*スケーリング* P. 付録 I-16

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-1 Pseudocode operators and keywords (continued)

Operator	Meaning	See
AND	Bitwise AND of bitstrings	*Logical operations on bitstrings* on page AppxI-13
array	Keyword introducing array type definition	*Arrays* on page AppxI-8
bit	Bitstring type of length 1	*Bitstrings* on page AppxI-5
bits(N)	Bitstring type of length N	*Bitstrings* on page AppxI-5
boolean	Boolean type	*Booleans* on page AppxI-6
case ... of ...	Control structure	*case ... of ...* on page AppxI-20
DIV	Quotient from integer division	*Division and modulo* on page AppxI-15
enumeration	Keyword introducing enumeration type definition	*Enumerations* on page AppxI-6
EOR	Bitwise EOR of bitstrings	*Logical operations on bitstrings* on page AppxI-13
FALSE	Boolean constant	*Booleans* on page AppxI-6
for ...	Control structure	*for ...* on page AppxI-20
if ... then ... else ...	Expression selecting between two values	*Conditional selection* on page AppxI-11
if ... then ... else ...	Control structure	*if ... then ... else ...* on page AppxI-19
IMPLEMENTATION_DEFINED	Describes IMPLEMENTATION DEFINED behavior	*IMPLEMENTATION_DEFINED* on page AppxI-18
integer	Unbounded integer type	*Integers* on page AppxI-6
MOD	Remainder from integer division	*Division and modulo* on page AppxI-15
OR	Bitwise OR of bitstrings	*Logical operations on bitstrings* on page AppxI-13
otherwise	Introduces default case in case ... of ... control structure	*case ... of ...* on page AppxI-20
real	Real number type	*Reals* on page AppxI-6
repeat ... until ...	Control structure	*repeat ... until ...* on page AppxI-20

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 J-1 擬似コードの演算子とキーワード （続き）

演算子	意味	参照先
AND	ビットストリングのビット単位論理積	*ビットストリングにおける論理演算* P. 付録 I-13
array	配列型の定義を開始するキーワード	*配列* P. 付録 I-8
bit	長さ 1 のビットストリング型	*ビットストリング* P. 付録 I-5
bits(N)	長さ N のビットストリング型	*ビットストリング* P. 付録 I-5
boolean	ブール型	*ブール* P. 付録 I-6
case ... of ...	制御構造	*case ... of ...* P. 付録 I-20
DIV	整数除算の商	*除算とモジュロ* P. 付録 I-15
enumeration	列挙型の定義を開始するキーワード	*列挙型* P. 付録 I-6
EOR	ビットストリングのビット単位排他的論理和	*ビットストリングにおける論理演算* P. 付録 I-13
FALSE	ブール定数	*ブール* P. 付録 I-6
for ...	制御構造	*for ...* P. 付録 I-20
if ... then ... else ...	2 つの値のいずれかを選択する式	*条件付き選択* P. 付録 I-11
if ... then ... else ...	制御構造	*if ... then ... else ...* P. 付録 I-19
IMPLEMENTATION_DEFINED	実装定義の動作の説明	*IMPLEMENTATION_DEFINED* P. 付録 I-18
integer	制限なしの整数型	*整数* P. 付録 I-6
MOD	整数除算の剰余	*除算とモジュロ* P. 付録 I-15
OR	ビットストリングのビット単位論理和	*ビットストリングにおける論理演算* P. 付録 I-13
otherwise	case ... of ... 制御構造のデフォルトケースの開始	*case ... of ...* P. 付録 I-20
real	実数型	*実数* P. 付録 I-6
repeat ... until ...	制御構造	*repeat ... until ...* P. 付録 I-20

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-1 Pseudocode operators and keywords (continued)

Operator	Meaning	See
`return`	Procedure or function return	*Return statements* on page AppxI-17
`SEE`	Points to other pseudocode to use instead	*SEE...* on page AppxI-18
`SUBARCHITECTURE_DEFINED`	Describes SUBARCHITECTURE DEFINED behavior	*SUBARCHITECTURE_DEFINED* on page AppxI-18
`TRUE`	Boolean constant	*Booleans* on page AppxI-6
`UNDEFINED`	Cause Undefined Instruction exception	*UNDEFINED* on page AppxI-17
`UNKNOWN`	Unspecified value	*General expression syntax* on page AppxI-9
`UNPREDICTABLE`	Unspecified behavior	*UNPREDICTABLE* on page AppxI-18
`when`	Introduces specific case in `case ... of ...` control structure	*case ... of ...* on page AppxI-20
`while ... do ...`	Control structure	*while ... do* on page AppxI-20

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 J-1 擬似コードの演算子とキーワード （続き）

演算子	意味	参照先
`return`	プロシージャまたは関数からの復帰	*リターン文* P. 付録 I-17
`SEE`	代わりに使用する他の擬似コードを指し示す	*SEE...* P. 付録 I-18
`SUBARCHITECTURE_DEFINED`	サブアーキテクチャ定義の動作を説明する	*SUBARCHITECTURE_DEFINED* P. 付録 I-18
`TRUE`	ブール定数	*ブール* P. 付録 I-6
`UNDEFINED`	未定義命令例外を引き起こす	*未定義* P. 付録 I-17
`UNKNOWN`	指定されていない値	*一般的な式の構文* P. 付録 I-9
`UNPREDICTABLE`	指定されていない動作	*予測不能* P. 付録 I-18
`when`	`case ... of ...` 制御構造の特定のケースを開始する	*case ... of ...* P. 付録 I-20
`while ... do ...`	制御構造	*while ... do* P. 付録 I-20

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

J.2 Pseudocode functions and procedures

Table J-2 lists the pseudocode functions and procedures used in this manual, and is an index to their descriptions:

Table J-2 Pseudocode functions and procedures

Function	Meaning	See
`_Mem[]`	Basic memory accesses	*Basic memory accesses* on page B2-30
`Abs()`	Absolute value of an integer or real	*Unary plus, minus and absolute value* on page AppxI-15
`AddWithCarry()`	Addition of bitstrings, with carry input and carry/overflow outputs	*Pseudocode details of addition and subtraction* on page A2-8
`AdvancedSIMDExpandImm()`	Expansion of immediates for Advanced SIMD instructions	*Operation* on page A7-23
`Align()`	Align integer or bitstring to multiple of an integer	*Rounding and aligning* on page AppxI-16
`AlignmentFault()`	Generate an Alignment fault on the memory system in use	*Interfaces to memory system specific pseudocode* on page B2-30
`AlignmentFaultP()`	Generate an Alignment fault on the PMSA memory system	*Alignment fault* on page B4-79
`AlignmentFaultV()`	Generate an Alignment fault on the VMSA memory system	*Alignment fault* on page B3-156
`ALUWritePC()`	Write value to PC, with interworking for ARM only from ARMv7	*Pseudocode details of operations on ARM core registers* on page A2-12
`ArchVersion()`	Major version number of the architecture	*ArchVersion()* on page AppxI-22
`ARMExpandImm()`	Expansion of immediates for ARM instructions	*Operation* on page A5-10
`ARMExpandImm_C()`	Expansion of immediates for ARM instructions, with carry output	
`ASR()`	Arithmetic shift right of a bitstring	*Shift and rotate operations* on page A2-5
`ASR_C()`	Arithmetic shift right of a bitstring, with carry output	
`AssignToTLB()`	Allocate new TLB entry	*TLB operations* on page B3-158

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

J.2 擬似コードの関数とプロシージャ

表 J-2 は、このマニュアルで使用されている擬似コードの関数とプロシージャを一覧表記し、それぞれの説明のあるページを示しています。

表 J-2 擬似コードの関数とプロシージャ

機能	意味	参照先
`_Mem[]`	基本的なメモリアクセス	*基本的なメモリアクセス* P. B2-30
`Abs()`	整数または実数の絶対値	*単項のプラス、マイナス、絶対値* P. 付録 I-14
`AddWithCarry()`	キャリー入力とキャリー / オーバフロー出力を伴うビットストリングの加算	*加算および減算の擬似コードの詳細* P. A2-8
`AdvancedSIMDExpandImm()`	アドバンスト SIMD 命令におけるイミディエート値の拡張	*動作* P. A7-23
`Align()`	整数の倍数に整数またはビットストリングをアラインする	*丸めとアライン* P. 付録 I-16
`AlignmentFault()`	使用中のメモリシステムでアライメントフォールトを生成する	*メモリシステム固有の擬似コードに対するインタフェース* P. B2-30
`AlignmentFaultP()`	PMSA メモリシステムでアライメントフォールトを生成する	*アライメントフォールト* P. B4-79
`AlignmentFaultV()`	VMSA メモリシステムでアライメントフォールトを生成する	*アライメントフォールト* P. B3-156
`ALUWritePC()`	PC に値を書き込み、ARMv7 以降のみ ARM のインターワーキング	*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12
`ArchVersion()`	アーキテクチャのメジャーバージョン番号	*ArchVersion()* P. 付録 I-22
`ARMExpandImm()`	ARM 命令におけるイミディエート値の拡張	*動作* P. A5-10
`ARMExpandImm_C()`	ARM 命令におけるイミディエート値の拡張、キャリー出力付き	
`ASR()`	ビットストリングの算術右シフト	*シフトおよびローテート演算* P. A2-5
`ASR_C()`	ビットストリングの算術右シフト、キャリー出力付き	
`AssignToTLB()`	新しい TLB エントリを割り当てる	*TLB 操作* P. B3-158

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`BadMode()`	Test whether mode number is valid	*Pseudocode details of mode operations* on page B1-8
`BadReg()`	Test for register number 13 or 15	*BadReg()* on page AppxI-22
`BigEndian()`	Returns TRUE if big-endian memory accesses selected	*ENDIANSTATE* on page A2-19
`BigEndianReverse()`	Endian-reverse the bytes of a bitstring	*Reverse endianness* on page B2-34
`BitCount()`	Count number of ones in a bitstring	*Bitstring count* on page AppxI-13
`BKPTInstrDebugEvent()`	Generate a debug event for a BKPT instruction	*Debug events* on page C3-27
`BranchTo()`	Continue execution at specified address	*Pseudocode details of ARM core register operations* on page B1-12
`BranchWritePC()`	Write value to PC, without interworking	*Pseudocode details of operations on ARM core registers* on page A2-12
`BreakpointDebugEvent()`	Generate a debug event for a breakpoint	*Debug events* on page C3-27
`BRPLinkMatch()`	Check whether an access matches a linked Breakpoint Register Pair	*Breakpoints and Vector Catches* on page C3-28
`BRPMatch()`	Check whether an instruction unit access matches a Breakpoint Register Pair	
`BXWritePC()`	Write value to PC, with interworking	*Pseudocode details of operations on ARM core registers* on page A2-12
`CallSupervisor()`	Generate exception for SVC instruction	*CallSupervisor()* on page AppxI-22
`CheckAdvSIMDEnabled()`	Undefined Instruction exception if the Advanced SIMD extension is not enabled	*Pseudocode details of enabling the Advanced SIMD and VFP extensions* on page B1-65
`CheckAdvSIMDOrVFPEnabled()`	Undefined Instruction exception if the specified one of the Advanced SIMD and VFP extensions is not enabled	*Pseudocode details of enabling the Advanced SIMD and VFP extensions* on page B1-65
`CheckDomain()`	VMSA check for Domain fault	*Domain checking* on page B3-157
`CheckPermissions()`	Memory system check of access permissions	*Access permission checking* on page B2-37

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
`BadMode()`	モード番号が有効かどうかをテストする	*モード動作の擬似コードの詳細* P. B1-8
`BadReg()`	レジスタ番号が 13 または 15 かどうかをテストする	*BadReg()* P. 付録 I-22
`BigEndian()`	ビッグエンディアンのメモリアクセスが選択されている場合に TRUE を返す	*ENDIANSTATE* P. A2-19
`BigEndianReverse()`	ビットストリングのバイトのエンディアン反転	*エンディアン形式の反転* P. B2-34
`BitCount()`	ビットストリング内の 1 の数をカウントする	*ビットストリングのカウント* P. 付録 I-13
`BKPTInstrDebugEvent()`	BKPT 命令のデバッグイベントを生成する	*デバッグイベント* P. C3-27
`BranchTo()`	指定したアドレスで実行を継続する	*ARM コアレジスタ演算の擬似コードの詳細* P. B1-12
`BranchWritePC()`	PC に値を書き込み、インターワーキングは行わない	*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12
`BreakpointDebugEvent()`	ブレークポイントのためのデバッグイベントを生成する	*デバッグイベント* P. C3-27
`BRPLinkMatch()`	アクセスがリンクしたブレークポイントレジスタペアに一致するかどうかをチェックする	*ブレークポイントとベクタキャッチ* P. C3-28
`BRPMatch()`	命令ユニットアクセスがブレークポイント レジスタペアに一致するかどうかをチェックする	
`BXWritePC()`	PC に値を書き込み、インターワーキングを行う	*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12
`CallSupervisor()`	SVC 命令のための例外を生成する	*CallSupervisor()* P. 付録 I-22
`CheckAdvSIMDEnabled()`	アドバンスト SIMD 拡張機能が稼動していない場合は未定義命令例外	*アドバンスト SIMD 拡張機能と VFP 拡張機能を許可する擬似コードの詳細* P. B1-65

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`CheckTLB()`	Check whether TLB entry exists for an address	*TLB operations* on page B3-158
`CheckVFPEnabled()`	Undefined Instruction exception if the VFP extension is not enabled	*Pseudocode details of enabling the Advanced SIMD and VFP extensions* on page B1-65
`ClearEventRegister()`	Clear the Event Register of the current processor	*Pseudocode details of the Wait For Event lock mechanism* on page B1-46
`ClearExclusiveByAddress()`	Clear global exclusive monitor records for an address range	*Exclusive monitors operations* on page B2-35
`ClearExclusiveLocal()`	Clear local exclusive monitor record of a processor	
`ConditionPassed()`	Returns TRUE if the current instruction passes its condition check	*Pseudocode details of conditional execution* on page A8-9
`Consistent()`	Test identically-named instruction bits or fields are identical	*Instruction encoding diagrams and pseudocode* on page AppxI-2
`Coproc_Accepted()`	Determine whether a coprocessor accepts an instruction	*Coproc_Accepted()* on page AppxI-22
`Coproc_DoneLoading()`	Returns TRUE if enough words have been loaded, for an LDC or LDC2 instruction	*Coproc_DoneLoading()* on page AppxI-23
`Coproc_DoneStoring()`	Returns TRUE if enough words have been stored, for an STC or STC2 instruction	*Coproc_DoneStoring()* on page AppxI-23
`Coproc_GetOneWord()`	Get word from coprocessor, for an MRC or MRC2 instruction	*Coproc_GetOneWord()* on page AppxI-23
`Coproc_GetTwoWords()`	Get two words from coprocessor, for an MRRC or MRRC2 instruction	*Coproc_GetTwoWords()* on page AppxI-23
`Coproc_GetWordToStore()`	Get next word to store from coprocessor, for STC or STC2 instruction	*Coproc_GetWordToStore()* on page AppxI-23
`Coproc_InternalOperation()`	Instruct coprocessor to perform an internal operation, for a CDP or CDP2 instruction	*Coproc_InternalOperation()* on page AppxI-23

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
`CheckAdvSIMDOrVFPEnabled()`	指定したアドバンスト SIMD 拡張機能または VFP 拡張機能が稼動していない場合は未定義命令例外	*アドバンスト SIMD 拡張機能と VFP 拡張機能を許可する擬似コードの詳細* P. B1-65
`CheckDomain()`	ドメインフォールトについて VMSA をチェックする	*ドメインチェック* P. B3-157
`CheckPermissions()`	アクセス許可についてメモリシステムをチェックする	*アクセス許可のチェック* P. B2-37
`CheckTLB()`	アドレスの TLB エントリが存在するかどうかチェックする	*TLB 操作* P. B3-158
`CheckVFPEnabled()`	VFP 拡張機能が稼動していない場合は未定義命令例外	*アドバンスト SIMD 拡張機能と VFP 拡張機能を許可する擬似コードの詳細* P. B1-65
`ClearEventRegister()`	現在のプロセッサのイベントレジスタをクリアする	*イベント待ちロック機構の擬似コードの詳細* P. B1-46
`ClearExclusiveByAddress()`	アドレス範囲のグローバル排他モニタレコードをクリアする	*排他モニタ操作* P. B2-35
`ClearExclusiveLocal()`	プロセッサのローカル排他モニタレコードをクリアする	
`ConditionPassed()`	現在の命令が条件チェックにパスする場合に TRUE を返す	*条件付き実行の擬似コードの詳細* P. A8-9
`Consistent()`	同一の名前を持つ命令ビットやフィールドが同一かどうかをテストする	*命令エンコード図および擬似コード* P. 付録 I-2
`Coproc_Accepted()`	コプロセッサが命令を受け付けるかどうかを判定する	*Coproc_Accepted()* P. 付録 I-22
`Coproc_DoneLoading()`	LDC または LDC2 命令で十分な数のワードがロードされている場合は TRUE を返す	*Coproc_DoneLoading()* P. 付録 I-23
`Coproc_DoneStoring()`	STC または STC2 命令で十分な数のワードがストアされている場合は TRUE を返す	*Coproc_DoneStoring()* P. 付録 I-23
`Coproc_GetOneWord()`	MRC または MRC2 命令のためのワードをコプロセッサから取得する	*Coproc_GetOneWord()* P. 付録 I-23

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`Coproc_SendLoadedWord()`	Send next loaded word to coprocessor, for LDC or LDC2 instruction	*Coproc_SendLoadedWord()* on page AppxI-23
`Coproc_SendOneWord()`	Send word to coprocessor, for an MCR or MCR2 instruction	*Coproc_SendOneWord()* on page AppxI-23
`Coproc_SendTwoWords()`	Send two words to coprocessor, for an MCRR or MCRR2 instruction	*Coproc_SendTwoWords()* on page AppxI-24
`CountLeadingSignBits()`	Number of identical sign bits at left end of bitstring, excluding the leftmost bit itself	*Lowest and highest set bits of a bitstring* on page AppxI-13
`CountLeadingZeroBits()`	Number of zeros at left end of bitstring	
`CPSRWriteByInstr()`	CPSR write by an instruction	*Pseudocode details of PSR operations* on page B1-20
`CurrentCond()`	Returns condition for current instruction	*Pseudocode details of conditional execution* on page A8-9
`CurrentInstrSet()`	Returns the instruction set currently in use	*ISETSTATE* on page A2-15
`CurrentModeIsPrivileged()`	Returns TRUE if current mode is privileged	*Pseudocode details of mode operations* on page B1-8
`CurrentModeIsUserOrSystem()`	Returns TRUE if current mode is User or System mode	
`D[]`	Doubleword / double-precision view of the Advanced SIMD and VFP registers	*Advanced SIMD and VFP extension registers* on page A2-21
`DataAbort()`	Cause a Data Abort exception of a specified type	*Data Abort exception* on page B2-39
`DataMemoryBarrier()`	Perform a Data Memory Barrier operation	*Pseudocode details of memory barriers* on page A3-50
`DataSynchronizationBarrier()`	Perform a Data Synchronization Barrier operation	
`Debug_CheckDataAccess()`	Check a data access for watchpoints	*Watchpoints* on page C3-35
`Debug_CheckInstruction()`	Check an instruction access for breakpoints and vector catches	*Breakpoints and Vector Catches* on page C3-28

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
`Coproc_GetTwoWords()`	MRRC または MRRC2 命令のための 2 ワードをコプロセッサから取得する	*Coproc_GetTwoWords()* P. 付録 I-23
`Coproc_GetWordToStore()`	STC または STC2 命令でストアする次のワードをコプロセッサから取得する	*Coproc_GetWordToStore()* P. 付録 I-23
`Coproc_InternalOperation()`	CDP または CDP2 命令について内部操作を実行するようコプロセッサに指示する	*Coproc_InternalOperation()* P. 付録 I-23
`Coproc_SendLoadedWord()`	LDC または LDC2 命令のためにロードされた次のワードをコプロセッサに送信する	*Coproc_SendLoadedWord()* P. 付録 I-23
`Coproc_SendOneWord()`	MRC または MRC2 命令のワードをコプロセッサに送信する	*Coproc_SendOneWord()* P. 付録 I-23
`Coproc_SendTwoWords()`	MCRR または MCRR2 命令のための 2 ワードをコプロセッサに送信する	*Coproc_SendTwoWords()* P. 付録 I-24
`CountLeadingSignBits()`	ビットストリングの左端にある同一の符号ビットの数、ただし一番左にあるビットを除く	*ビットストリングでセットされている最下位および最上位ビット* P. 付録 I-13
`CountLeadingZeroBits()`	ビットストリングの左端にある 0 の数	
`CPSRWriteByInstr()`	命令による CPSR 書き込み	*PSR 動作の擬似コードの詳細* P. B1-20
`CurrentCond()`	現在の命令の条件を返す	*条件付き実行の擬似コードの詳細* P. A8-9
`CurrentInstrSet()`	現在使用されている命令セットを返す	*ISETSTATE* P. A2-15
`CurrentModeIsPrivileged()`	現在のモードが特権モードの場合に TRUE を返す	*モード動作の擬似コードの詳細* P. B1-8
`CurrentModeIsUserOrSystem()`	現在のモードがユーザモードまたはシステムモードの場合に TRUE を返す	
`D[]`	アドバンスト SIMD および VFP レジスタのダブルワード / 倍精度ビュー	*アドバンスト SIMD と VFP の拡張機能レジスタ* P. A2-21
`DataAbort()`	指定したタイプのデータアボート例外を引き起こす	*データアボート例外* P. B2-39

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`DecodeImmShift()`	Decode shift type and amount for an immediate shift	*Pseudocode details of instruction-specified shifts and rotates* on page A8-11
`DecodeRegShift()`	Decode shift type for a register-controlled shift	
`DefaultAttrs()`	Determine memory attributes for an address in the PMSA default memory map	*Default memory map attributes* on page B4-81
`DefaultTEXDecode()`	Determine default memory attributes for a set of TEX[2:0], C, B bits	*Default memory access decode* on page B2-37
`Elem[]`	Access element of a vector	*Advanced SIMD vectors* on page A2-26
`EncodingSpecificOperations()`	Invoke encoding-specific pseudocode and *should be* checks	*Instruction encoding diagrams and pseudocode* on page AppxI-2
`EndOfInstruction()`	Terminate processing of current instruction	*EndOfInstruction()* on page AppxI-24
`EventRegistered()`	Determine whether the Event Register of the current processor is set	*Pseudocode details of the Wait For Event lock mechanism* on page B1-46
`ExclusiveMonitorsPass()`	Check whether Store-Exclusive operation has control of exclusive monitors	*Exclusive monitors operations* on page B2-35
`ExcVectorBase()`	Return non-Monitor mode exception base address for current security state	*Operation* on page B1-33
`FixedToFP()`	Convert integer or fixed-point to floating-point	*Conversions* on page A2-64
`FPAbs()`	Floating-point absolute value	*Negation and absolute value* on page A2-47
`FPAdd()`	Floating-point addition	*Addition and subtraction* on page A2-55
`FPCompare()`	Floating-point comparison, producing NZCV flag result	*Comparisons* on page A2-53
`FPCompareEQ()`	Floating-point test for equality	

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
`DataMemoryBarrier()`	データメモリ バリア操作を実行する	*メモリバリアの擬似コードの詳細* P. A3-50
`DataSynchronizationBarrier()`	データ同期バリア操作を実行する	
`Debug_CheckDataAccess()`	ウォッチポイントについてデータアクセスをチェックする	*ウォッチポイント* P. C3-35
`Debug_CheckInstruction()`	ブレークポイントおよびベクタキャッチについて命令アクセスをチェックする	*ブレークポイントとベクタキャッチ* P. C3-28
`DecodeImmShift()`	イミディエートシフトのシフトタイプと量をデコードする	*命令により指定されるシフトとローテートの擬似コードの詳細* P. A8-11
`DecodeRegShift()`	レジスタ制御シフトのシフトタイプをデコードする	
`DefaultAttrs()`	PMSA デフォルト メモリマップにあるアドレスのメモリ属性を判定する	*デフォルトのメモリマップ属性* P. B4-81
`DefaultTEXDecode()`	TEX[2:0]、C、B ビットのセットについてデフォルトのメモリ属性を判定する	*デフォルトのメモリアクセス デコード* P. B2-37
`Elem[]`	ベクタのエレメントにアクセスする	*アドバンスト SIMD のベクタ* P. A2-26
`EncodingSpecificOperations()`	エンコード固有の擬似コードと *should be* チェックを呼び出す	*命令エンコード図および擬似コード* P. 付録 I-2
`EndOfInstruction()`	現在の命令の処理を終了する	*EndOfInstruction()* P. 付録 I-24
`EventRegistered()`	現在のプロセッサのイベントレジスタがセットされているかどうかを判定する	*イベント待ちロック機構の擬似コードの詳細* P. B1-46
`ExclusiveMonitorsPass()`	排他ストア操作が排他モニタの制御を持つかどうかをチェックする	*排他モニタ操作* P. B2-35
`ExcVectorBase()`	現在のセキュリティ状態について非モニタモード例外のベースアドレスを返す	*動作* P. B1-33
`FixedToFP()`	整数または固定小数点数を浮動小数点数に変換する	*変換* P. A2-64
`FPAbs()`	浮動小数点数の絶対値	*否定と絶対値* P. A2-47
`FPAdd()`	浮動小数点数の加算	*加算と減算* P. A2-55

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`FPCompareGE()`	Floating-point test for greater than or equal	*Comparisons* on page A2-53
`FPCompareGT()`	Floating-point test for greater than	
`FPDefaultNaN()`	Generate floating-point default NaN	*Generation of specific floating-point values* on page A2-46
`FPDiv()`	Floating-point division	*Multiplication and division* on page A2-57
`FPDoubleToSingle()`	Convert double-precision floating-point to single-precision floating-point	*Conversions* on page A2-64
`FPHalfToSingle()`	Convert half-precision floating-point to single-precision floating-point	
`FPInfinity()`	Generate floating-point infinity	*Generation of specific floating-point values* on page A2-46
`FPMax()`	Floating-point maximum	*Maximum and minimum* on page A2-55
`FPMaxNormal()`	Generate maximum normalized floating-point value	*Generation of specific floating-point values* on page A2-46
`FPMin()`	Floating-point minimum	*Maximum and minimum* on page A2-55
`FPMul()`	Floating-point multiplication	*Multiplication and division* on page A2-57
`FPNeg()`	Floating-point negation	*Negation and absolute value* on page A2-47
`FPProcessException()`	Process a floating-point exception	*Floating-point exception and NaN handling* on page A2-49
`FPProcessNaN()`	Generate correct result and exceptions for a NaN operand	
`FPProcessNaNs()`	Perform NaN operand checks and processing for a 2-operand floating-point operation	

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
`FPCompare()`	浮動小数点数を比較し、NZCV フラグの結果を生成する	*比較* P. A2-53
`FPCompareEQ()`	浮動小数点数が等しいかどうかをテストする	
`FPCompareGE()`	浮動小数点数の左辺が右辺より大きいまたは等しいかどうかをテストする	*比較* P. A2-53
`FPCompareGT()`	浮動小数点数の左辺が右辺より大きいかどうかをテストする	
`FPDefaultNaN()`	浮動小数点のデフォルト NaN を生成する	*特定の浮動小数点値の生成* P. A2-46
`FPDiv()`	浮動小数点数の除算	*乗算と除算* P. A2-57
`FPDoubleToSingle()`	倍精度の浮動小数点数から単精度の浮動小数点数への変換	*変換* P. A2-64
`FPHalfToSingle()`	半精度の浮動小数点数から単精度の浮動小数点数への変換	
`FPInfinity()`	浮動小数点の無限大を生成する	*特定の浮動小数点値の生成* P. A2-46
`FPMax()`	浮動小数点数の最大値	*最大値と最小値* P. A2-55
`FPMaxNormal()`	最大の正規化浮動小数点数値を生成する	*特定の浮動小数点値の生成* P. A2-46
`FPMin()`	浮動小数点数の最小値	*最大値と最小値* P. A2-55
`FPMul()`	浮動小数点数の乗算	*乗算と除算* P. A2-57
`FPNeg()`	浮動小数点数の否定	*否定と絶対値* P. A2-47
`FPProcessException()`	浮動小数点例外の処理	*浮動小数点例外と NaN の処理* P. A2-49
`FPProcessNaN()`	NaN オペランドに対して正しい結果と例外を生成する	
`FPProcessNaNs()`	NaN オペランドチェックおよび 2 オペランド浮動小数点演算の処理を実行する	

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
FPRecipEstimate()	Floating-point reciprocal estimate	*Reciprocal estimate and step* on page A2-58
FPRecipStep()	Floating-point 2-xy operation for Newton-Raphson reciprocal iteration	
FPRound()	Floating-point rounding	*Floating-point rounding* on page A2-51
FPSingleToDouble()	Convert single-precision floating-point to double-precision floating-point	*Conversions* on page A2-64
FPSingleToHalf()	Convert single-precision floating-point to half-precision floating-point	
FPRSqrtEstimate()	Floating-point reciprocal square root estimate	*Reciprocal square root* on page A2-61
FPRSqrtStep()	Floating-point (3-xy)/2 operation for Newton-Raphson reciprocal square root iteration	
FPSqrt()	Floating-point square root	*Square root* on page A2-60
FPSub()	Floating-point subtraction	*Addition and subtraction* on page A2-55
FPThree()	Generate floating-point value 3.0	*Generation of specific floating-point values* on page A2-46
FPToFixed()	Convert floating-point to integer or fixed-point	*Conversions* on page A2-64
FPTwo()	Generate floating-point value 2.0	*Generation of specific floating-point values* on page A2-46
FPUnpack()	Produce type, sign bit and real value of a floating-point number	*Floating-point value unpacking* on page A2-48
FPZero()	Generate floating-point zero	*Generation of specific floating-point values* on page A2-46
GenerateAlignmentException()	Generate the exception for a failed address alignment check	*GenerateAlignmentException()* on page AppxI-24
GenerateCoprocessorException()	Generate the exception for an unclaimed coprocessor instruction	*GenerateCoprocessorException()* on page AppxI-24

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
FPRecipEstimate()	浮動小数点の逆数概算	*逆数概算とステップ* P. A2-58
FPRecipStep()	ニュートン・ラプソン反復法で逆数を計算するための浮動小数点の 2-xy 演算	
FPRound()	浮動小数点数の丸め	*浮動小数点数の丸め* P. A2-51
FPSingleToDouble()	単精度の浮動小数点数から倍精度の浮動小数点数への変換	*変換* P. A2-64
FPSingleToHalf()	単精度の浮動小数点数から半精度の浮動小数点数への変換	
FPRSqrtEstimate()	浮動小数点の逆数平方根の概算	*逆数平方根* P. A2-61
FPRSqrtStep()	ニュートン・ラプソン反復法で逆数平方根を計算するための浮動小数点の (3-xy)/2 演算	
FPSqrt()	浮動小数点の平方根	*平方根* P. A2-60
FPSub()	浮動小数点数の減算	*加算と減算* P. A2-55
FPThree()	浮動小数点数値 3.0 を生成する	*特定の浮動小数点値の生成* P. A2-46
FPToFixed()	浮動小数点数を整数または固定小数点数に変換する	*変換* P. A2-64
FPTwo()	浮動小数点数値 2.0 を生成する	*特定の浮動小数点値の生成* P. A2-46
FPUnpack()	浮動小数点数の型、符号ビット、実数値を生成する	*浮動小数点値のアンパッキング* P. A2-48
FPZero()	浮動小数点の 0 を生成する	*特定の浮動小数点値の生成* P. A2-46
GenerateAlignmentException()	失敗したアドレス アライメントチェックのための例外を生成する	*GenerateAlignmentException()* P. 付録 I-24
GenerateCoprocessorException()	要求されていないコプロセッサ命令のための例外を生成する	*GenerateCoprocessorException()* P. 付録 I-24

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`GenerateIntegerZeroDivide()`	Generate the exception for a trapped divide-by-zero for an integer divide instruction	*GenerateIntegerZeroDivide()* on page AppxI-24
`HaveMPExt()`	Returns TRUE if the MP Extensions are implemented	*HaveMPExt()* on page AppxI-24
`HaveSecurityExt()`	Returns TRUE if the Security Extensions are implemented	*Pseudocode details of Secure state operations* on page B1-28
`HighestSetBit()`	Position of leftmost 1 in a bitstring	*Lowest and highest set bits of a bitstring* on page AppxI-13
`Hint_Debug()`	Perform function of DBG hint instruction	*Hint_Debug()* on page AppxI-24
`Hint_PreloadData()`	Perform function of PLD memory hint instruction	*Hint_PreloadData()* on page AppxI-25
`Hint_PreloadDataForWrite()`	Perform function of PLDW Memory hint instruction	*Hint_PreloadDataForWrite()* on page AppxI-25
`Hint_PreloadInstr()`	Perform function of PLI memory hint instruction	*Hint_PreloadInstr()* on page AppxI-25
`Hint_Yield()`	Perform function of YIELD hint instruction	*Hint_Yield()* on page AppxI-25
`InITBlock()`	Return TRUE if current instruction is in an IT block	*ITSTATE* on page A2-17
`InstructionSynchronizationBarrier()`	Perform an Instruction Synchronization Barrier operation	*Pseudocode details of memory barriers* on page A3-50
`Int()`	Convert bitstring to integer in argument-specified fashion	*Converting bitstrings to integers* on page AppxI-14
`IntegerZeroDivideTrappingEnabled()`	Check whether divide-by-zero trapping is enabled for integer divide instructions	*IntegerZeroDivideTrappingEnabled()* on page AppxI-25
`IsExclusiveGlobal()`	Check a global exclusive access record	*Exclusive monitors operations* on page B2-35
`IsExclusiveLocal()`	Check a local exclusive access record	
`IsExternalAbort()`	Returns TRUE if abort being processed is an external abort	*IsExternalAbort()* on page AppxI-25

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
`GenerateIntegerZeroDivide()`	整数除算命令について、トラップされた 0 による除算のための例外を生成する	*GenerateIntegerZeroDivide()* P. 付録 I-24
`HaveMPExt()`	MP 拡張機能が実装されている場合は TRUE を返す	*HaveMPExt()* P. 付録 I-24
`HaveSecurityExt()`	セキュリティ拡張機能が実装されている場合は TRUE を返す	*セキュア状態動作の擬似コードの詳細* P. B1-28
`HighestSetBit()`	ビットストリングにおいて最も左にある 1 の位置	*ビットストリングでセットされている最下位および最上位ビット* P. 付録 I-13
`Hint_Debug()`	DBG ヒント命令の関数を実行する	*Hint_Debug()* P. 付録 I-24
`Hint_PreloadData()`	PLD メモリヒント命令の関数を実行する	*Hint_PreloadData()* P. 付録 I-25
`Hint_PreloadDataForWrite()`	PLDW メモリヒント命令の関数を実行する	*Hint_PreloadDataForWrite()* P. 付録 I-25
`Hint_PreloadInstr()`	PLI メモリヒント命令の関数を実行する	*Hint_PreloadInstr()* P. 付録 I-25
`Hint_Yield()`	YIELD ヒント命令の関数を実行する	*Hint_Yield()* P. 付録 I-25
`InITBlock()`	現在の命令が IT ブロック内にある場合は TRUE を返す	*ITSTATE* P. A2-17
`InstructionSynchronizationBarrier()`	命令同期バリア操作を実行する	*メモリバリアの擬似コードの詳細* P. A3-50
`Int()`	引数により指定される方法でビットストリングを整数に変換する	*ビットストリングの整数への変換* P. 付録 I-14
`IntegerZeroDivideTrappingEnabled()`	整数除算命令において 0 による除算のトラップが可能などうかをチェックする	*IntegerZeroDivideTrappingEnabled()* P. 付録 I-25
`IsExclusiveGlobal()`	グローバル排他アクセスレコードをチェックする	*排他モニタ操作* P. B2-35
`IsExclusiveLocal()`	ローカル排他アクセスレコードをチェックする	

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
IsOnes()	Test for all-ones bitstring (Boolean result)	*Testing a bitstring for being all zero or all ones* on page AppxI-13
IsOnesBit()	Test for all-ones bitstring (bit result)	
IsSecure()	Returns TRUE in Secure state or if no Security Extensions	*Pseudocode details of Secure state operations* on page B1-28
IsZero()	Test for all-zeros bitstring (Boolean result)	*Testing a bitstring for being all zero or all ones* on page AppxI-13
IsZeroBit()	Test for all-zeros bitstring (bit result)	
ITAdvance()	Advance the ITSTATE bits to their values for the next instruction	*ITSTATE* on page A2-17
JazelleAcceptsExecution()	Returns TRUE if the Jazelle extension can start bytecode execution	*JazelleAcceptsExecution()* on page AppxI-25
LastInITBlock()	Return TRUE if current instruction is the last instruction of an IT block	*ITSTATE* on page A2-17
Len()	Bitstring length	*Bitstring length and most significant bit* on page AppxI-12
LoadWritePC()	Write value to PC, with interworking (without it before ARMv5T)	*Pseudocode details of operations on ARM core registers* on page A2-12
LookUpRName()	Find banked register for specified register number and mode	*Pseudocode details of ARM core register operations* on page B1-12
LowestSetBit()	Position of rightmost 1 in a bitstring	*Lowest and highest set bits of a bitstring* on page AppxI-13
LSL()	Logical shift left of a bitstring	*Shift and rotate operations* on page A2-5
LSL_C()	Logical shift left of a bitstring, with carry output	
LSR()	Logical shift right of a bitstring	
LSR_C()	Logical shift right of a bitstring, with carry output	
MarkExclusiveGlobal()	Set a global exclusive access record	*Exclusive monitors operations* on page B2-35
MarkExclusiveLocal()	Set a local exclusive access record	

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
IsExternalAbort()	処理中のアボートが外部アボートの場合に TRUE を返す	*IsExternalAbort()* P. 付録 I-25
IsOnes()	すべて 1 のビットストリングかどうかをテストする（ブール型の結果）	*ビットストリングがすべて 0 またはすべて 1 であるかどうかのテスト* P. 付録 I-13
IsOnesBit()	すべて 1 のビットストリングかどうかをテストする（ビットの結果）	
IsSecure()	セキュア状態にあるか、セキュリティ拡張機能がない場合に TRUE を返す	*セキュア状態動作の擬似コードの詳細* P. B1-28
IsZero()	すべて 0 のビットストリングかどうかをテストする（ブール型の結果）	*ビットストリングがすべて 0 またはすべて 1 であるかどうかのテスト* P. 付録 I-13
IsZeroBit()	すべて 0 のビットストリングかどうかをテストする（ビットの結果）	
ITAdvance()	次の命令の値に ITSTATE ビットを進める	*ITSTATE* P. A2-17
JazelleAcceptsExecution()	Jazelle 拡張機能がバイトコード実行を開始できる場合は TRUE を返す	*JazelleAcceptsExecution()* P. 付録 I-25
LastInITBlock()	現在の命令が IT ブロックの最後の命令である場合は TRUE を返す	*ITSTATE* P. A2-17
Len()	ビットストリングの長さ	*ビットストリング長および最上位ビット* P. 付録 I-12
LoadWritePC()	PC に値を書き込み、インターワーキングを行う（ARMv5T 以前の場合はインターワーキングなし）	*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12
LookUpRName()	指定したレジスタ番号とモードについてバンクレジスタを検索する	*ARM コアレジスタ演算の擬似コードの詳細* P. B1-12
LowestSetBit()	ビットストリングで最も右にある 1 の位置	*ビットストリングでセットされている最下位および最上位ビット* P. 付録 I-13

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`Max()`	Maximum of integers or reals	*Maximum and minimum* on page AppxI-16
`MemA[]`	Memory access that must be aligned, at current privilege level	*Aligned memory accesses* on page B2-31
`MemA_unpriv[]`	Memory access that must be aligned, unprivileged	
`MemA_with_priv[]`	Memory access that must be aligned, at specified privilege level	
`MemorySystemArchitecture()`	Return memory architecture of system (VMSA or PMSA)	*MemorySystemArchitecture()* on page AppxI-26
`MemU[]`	Memory access without alignment requirement, at current privilege level	*Unaligned memory accesses* on page B2-32
`MemU_unpriv[]`	Memory access without alignment requirement, unprivileged	
`MemU_with_priv[]`	Memory access without alignment requirement, at specified privilege level	
`Min()`	Minimum of integers or reals	*Maximum and minimum* on page AppxI-16
`NOT()`	Bitwise inversion of a bitstring	*Logical operations on bitstrings* on page AppxI-13
`NullCheckIfThumbEE()`	Perform base register null check if a ThumbEE instruction	*Null checking* on page A9-3
`Ones()`	All-ones bitstring	*Bitstring concatenation and replication* on page AppxI-12
`PCStoreValue()`	Value stored when an ARM instruction stores the PC	*Pseudocode details of operations on ARM core registers* on page A2-12
`PolynomialMult()`	Multiplication of polynomials over {0, 1}	*Pseudocode details of polynomial multiplication* on page A2-67
`ProcessorID()`	Return integer identifying the processor	*ProcessorID()* on page AppxI-26
`Q[]`	Quadword view of the Advanced SIMD and VFP registers	*Advanced SIMD and VFP extension registers* on page A2-21

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
`LSL()`	ビットストリングの論理左シフト	*シフトおよびローテート演算* P. A2-5
`LSL_C()`	ビットストリングの論理左シフト、キャリー出力付き	
`LSR()`	ビットストリングの論理右シフト	
`LSR_C()`	ビットストリングの論理右シフト、キャリー出力付き	
`MarkExclusiveGlobal()`	グローバル排他アクセスレコードを設定する	*排他モニタ操作* P. B2-35
`MarkExclusiveLocal()`	ローカル排他アクセスレコードをセットする	
`Max()`	整数または実数の最大値	*最大値と最小値* P. 付録 I-16
`MemA[]`	アラインしなければならないメモリアクセス、現在の特権レベル	*アラインド メモリアクセス* P. B2-31
`MemA_unpriv[]`	アラインしなければならないメモリアクセス、非特権	
`MemA_with_priv[]`	アラインしなければならないメモリアクセス、指定された特権レベル	
`MemorySystemArchitecture()`	システムのメモリアーキテクチャを返す（VMSA または PMSA）	*MemorySystemArchitecture()* P. 付録 I-26
`MemU[]`	アラインする必要のないメモリアクセス、現在の特権レベル	*アンアラインドメモリアクセス* P. B2-32
`MemU_unpriv[]`	アラインする必要のないメモリアクセス、非特権	
`MemU_with_priv[]`	アラインする必要のないメモリアクセス、指定された特権レベル	
`Min()`	整数または実数の最小値	*最大値と最小値* P. 付録 I-16
`NOT()`	ビットストリングのビット単位の反転	*ビットストリングにおける論理演算* P. 付録 I-13

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
R[]	Access the main ARM core register bank, using current mode	*Pseudocode details of ARM core register operations* on page B1-12
RBankSelect()	Evaluate register banking for R13, R14	
RemapRegsHaveResetValues()	Check PRRR and NMRR for reset values	*RemapRegsHaveResetValues()* on page AppxI-26
Replicate()	Bitstring replication	*Bitstring concatenation and replication* on page AppxI-12
RfiqBankSelect()	Evaluate register banking for R8-R12	*Pseudocode details of ARM core register operations* on page B1-12
Rmode[]	Access the main ARM core register bank, using specified mode	
ROR()	Rotate right of a bitstring	*Shift and rotate operations* on page A2-5
ROR_C()	Rotate right of a bitstring, with carry output	
RoundDown()	Round real to integer (rounding towards –infinity)	*Rounding and aligning* on page AppxI-16
RoundTowardsZero()	Round real to integer (rounding towards zero)	
RoundUp()	Round real to integer (rounding towards +infinity)	
RRX()	Rotate right with extend of a bitstring	*Shift and rotate operations* on page A2-5
RRX_C()	Rotate right with extend of a bitstring, with carry output	
S[]	Single word / single-precision view of the Advanced SIMD and VFP registers	*Advanced SIMD and VFP extension registers* on page A2-21
Sat()	Convert integer to bitstring with specified saturation	*Pseudocode details of saturation* on page A2-9
SatQ()	Convert integer to bitstring with specified saturation, with saturated flag output	
SelectInstrSet()	Sets the instruction set currently in use	*ISETSTATE* on page A2-15

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
NullCheckIfThumbEE()	ThumbEE 命令の場合、ベースレジスタの NULL チェックを実行する	*NULL チェック* P. A9-3
Ones()	すべて 1 のビットストリング	*ビットストリングの連結と複製* P. 付録 I-12
PCStoreValue()	ARM 命令が PC をストアするときにストアされる値	*ARM コアレジスタ演算の擬似コードの詳細* P. A2-12
PolynomialMult()	{0, 1} における多項式の乗算	*多項式乗算の擬似コードの詳細* P. A2-67
ProcessorID()	プロセッサを識別する整数を返す	*ProcessorID()* P. 付録 I-26
Q[]	アドバンスト SIMD および VFP レジスタのクワッドワードビュー	*アドバンスト SIMD と VFP の拡張機能レジスタ* P. A2-21
R[]	現在のモードを使ってメインの ARM コア レジスタバンクにアクセスする	*ARM コアレジスタ演算の擬似コードの詳細* P. B1-12
RBankSelect()	R13、R14 のレジスタバンク処理を評価する	
RemapRegsHaveResetValues()	PRRR および NMRR のリセット時の値をチェックする	*RemapRegsHaveResetValues()* P. 付録 I-26
Replicate()	ビットストリングの複製	*ビットストリングの連結と複製* P. 付録 I-12
RfiqBankSelect()	R8 ～ R12 のレジスタバンク処理を評価する	*ARM コアレジスタ演算の擬似コードの詳細* P. B1-12
Rmode[]	指定されたモードを使ってメインの ARM コア レジスタバンクにアクセスする	
ROR()	ビットストリングの右ローテート	*シフトおよびローテート演算* P. A2-5
ROR_C()	ビットストリングの右ローテート、キャリー出力付き	
RoundDown()	実数を整数に丸める（- 無限大への丸め）	*丸めとアライン* P. 付録 I-16
RoundTowardsZero()	実数を整数に丸める（0 への丸め ）	
RoundUp()	実数を整数に丸める（+ 無限大への丸め）	

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
SendEvent()	Perform function of SEV hint instruction	*Pseudocode details of the Wait For Event lock mechanism* on page B1-46
SerializeVFP()	Ensure exceptional conditions in preceding VFP instructions have been detected	*Asynchronous bounces, serialization, and VFP exception barriers* on page B1-70
SetExclusiveMonitors()	Set exclusive monitors for a Load-Exclusive operation	*Exclusive monitors operations* on page B2-35
Shift()	Perform a specified shift by a specified amount on a bitstring	*Pseudocode details of instruction-specified shifts and rotates* on page A8-11
Shift_C()	Perform a specified shift by a specified amount on a bitstring, with carry output	
SignedSat()	Convert integer to bitstring with signed saturation	*Pseudocode details of saturation* on page A2-9
SignedSatQ()	Convert integer to bitstring with signed saturation, with saturated flag output	
SignExtend()	Extend bitstring to left with copies of its leftmost bit	*Zero-extension and sign-extension of bitstrings* on page AppxI-13
SInt()	Convert bitstring to integer in signed (two's complement) fashion	*Converting bitstrings to integers* on page AppxI-14
SPSR[]	Access the SPSR of the current mode	*Pseudocode details of PSR operations* on page B1-20
SPSRWriteByInstr()	SPSR write by an instruction	
SwitchToJazelleExecution()	Start Jazelle extension execution of bytecodes	*SwitchToJazelleExecution()* on page AppxI-26
TakeDataAbortException()	Perform a Data Abort exception entry	*Data Abort exception* on page B1-55
TakeFIQException()	Perform an FIQ interrupt exception entry	*FIQ exception* on page B1-60
TakeIRQException()	Perform an IRQ interrupt exception entry	*IRQ exception* on page B1-58
TakePrefetchAbortException()	Perform a Prefetch Abort exception entry	*Prefetch Abort exception* on page B1-54
TakeReset()	Perform a Reset exception entry	*Reset* on page B1-48

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
RRX()	ビットストリングの拡張付き右ローテート	*シフトおよびローテート演算* P. A2-5
RRX_C()	ビットストリングの拡張付き右ローテート、キャリー出力付き	
S[]	アドバンスト SIMD および VFP レジスタの単一ワード / 単精度ビュー	*アドバンスト SIMD と VFP の拡張機能レジスタ* P. A2-21
Sat()	指定の飽和で整数をビットストリングに変換する	*飽和の擬似コードの詳細* P. A2-9
SatQ()	指定の飽和で整数をビットストリングに変換する、飽和フラグの出力あり	
SelectInstrSet()	現在使用されている命令セットを設定する	*ISETSTATE* P. A2-15
SendEvent()	SEV ヒント命令の関数を実行する	*イベント待ちロック機構の擬似コードの詳細* P. B1-46
SerializeVFP()	先行する VFP 命令の例外条件が検出されたことを確認する	*非同期バウンス、直列化、VFP 例外バリア* P. B1-70
SetExclusiveMonitors()	排他ロード操作のための排他モニタをセットする	*排他モニタ操作* P. B2-35
Shift()	ビットストリングに対して、指定された量の指定のシフトを実行する	*命令により指定されるシフトとローテートの擬似コードの詳細* P. A8-11
Shift_C()	ビットストリングに対して、指定された量の指定のシフトを実行する、キャリー出力付き	
SignedSat()	符号付き飽和で整数をビットストリングに変換する	*飽和の擬似コードの詳細* P. A2-9
SignedSatQ()	符号付き飽和で整数をビットストリングに変換する、飽和フラグの出力あり	
SignExtend()	一番左にあるビットのコピーでビットストリングを左へ延長する	*ビットストリングのゼロ拡張および符号拡張* P. 付録 I-13
SInt()	符号付き（2 の補数）の形でビットストリングを整数に変換する	*ビットストリングの整数への変換* P. 付録 I-14

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`TakeSMCException()`	Perform a Secure Monitor Call exception entry	*Secure Monitor Call (SMC) exception* on page B1-53
`TakeSVCException()`	Perform a Supervisor Call exception entry	*Supervisor Call (SVC) exception* on page B1-52
`TakeUndefInstrException()`	Perform an Undefined Instruction exception entry	*Undefined Instruction exception* on page B1-49
`ThisInstr()`	Returns the bitstring encoding of the current instruction	*ThisInstr()* on page AppxI-26
`ThumbExpandImm()`	Expansion of immediates for Thumb instructions	*Operation* on page A6-18
`ThumbExpandImm_C()`	Expansion of immediates for Thumb instructions, with carry output	
`TopBit()`	Leftmost bit of a bitstring	*Bitstring length and most significant bit* on page AppxI-12
`TranslateAddress()`	Perform address translation and obtain memory attributes for a memory access	*Interfaces to memory system specific pseudocode* on page B2-30
`TranslateAddressP()`	Perform address translation and obtain memory attributes for a PMSA memory access	*Address translation* on page B4-79
`TranslateAddressV()`	Perform address translation and obtain memory attributes for a VMSA memory access	
`TranslateVAtoMVA()`	Fast Context Switch Extension virtual address to modified virtual address translation	*Modified virtual addresses* on page AppxE-3
`TranslationTableWalk()`	Perform VMSA translation table walk	*Translation table walk* on page B3-158
`UInt()`	Convert bitstring to integer in unsigned fashion	*Converting bitstrings to integers* on page AppxI-14
`UnalignedSupport()`	Check whether unaligned memory access support (introduced in ARMv6) is in use	*UnalignedSupport()* on page AppxI-26

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
`SPSR[]`	現在のモードの SPSR にアクセスする	*PSR 動作の擬似コードの詳細* P. B1-20
`SPSRWriteByInstr()`	命令による SPSR 書き込み	
`SwitchToJazelleExecution()`	バイトコードの Jazelle 拡張機能実行を開始する	*SwitchToJazelleExecution()* P. 付録 I-26
`TakeDataAbortException()`	データアボート例外開始を実行する	*データアボート例外* P. B1-55
`TakeFIQException()`	FIQ 割り込み例外開始を実行する	*FIQ 例外* P. B1-60
`TakeIRQException()`	IRQ 割り込み例外開始を実行する	*IRQ 例外* P. B1-58
`TakePrefetchAbortException()`	プリフェッチアボート例外開始を実行する	*プリフェッチアボート例外* P. B1-54
`TakeReset()`	リセット例外開始を実行する	*リセット* P. B1-48
`TakeSMCException()`	セキュアモニタ例外コール例外開始を実行する	*セキュア モニタコール (SMC) 例外* P. B1-53
`TakeSVCException()`	スーパバイザコール例外開始を実行する	*スーパバイザコール (SVC) 例外* P. B1-52
`TakeUndefInstrException()`	未定義命令例外開始を実行する	*未定義命令例外* P. B1-49
`ThisInstr()`	現在の命令のビットストリングエンコードを返す	*ThisInstr()* P. 付録 I-26
`ThumbExpandImm()`	Thumb 命令におけるイミディエート値の拡張	*動作* P. A6-18
`ThumbExpandImm_C()`	Thumb 命令におけるイミディエート値の拡張、キャリー出力付き	
`TopBit()`	ビットストリングの最も左にあるビット	*ビットストリング長および最上位ビット* P. 付録 I-12
`TranslateAddress()`	アドレス変換を実行し、メモリアクセスのためのメモリ属性を取得する	*メモリシステム固有の擬似コードに対するインタフェース* P. B2-30

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`UnsignedRecipEstimate()`	Unsigned fixed-point reciprocal estimate	*Reciprocal estimate and step* on page A2-58
`UnsignedRSqrtEstimate()`	Unsigned fixed-point reciprocal square root estimate	*Reciprocal square root* on page A2-61
`UnsignedSat()`	Convert integer to bitstring with unsigned saturation	*Pseudocode details of saturation* on page A2-9
`UnsignedSatQ()`	Convert integer to bitstring with unsigned saturation, with saturated flag output	
`VCR_OnTakingInterrupt()`	Track most recently used interrupt vectors for vector catch purposes	*Breakpoints and Vector Catches* on page C3-28
`VCRMatch()`	Check whether a vector catch occurs for an instruction unit access	
`VCRVectorMatch()`	Check whether an instruction unit access matches a vector	
`VectorCatchDebugEvent()`	Generate a debug event for a vector catch	*Debug events* on page C3-27
`VFPExcBarrier()`	Ensure all outstanding VFP exception processing has occurred	*Asynchronous bounces, serialization, and VFP exception barriers* on page B1-70
`VFPExpandImm()`	Expansion of immediates for VFP extension instructions	*Operation* on page A7-25
`VFPSmallRegisterBank()`	Returns TRUE if 16-doubleword VFP extension register bank implemented	*Pseudocode details of Advanced SIMD and VFP extension registers* on page A2-23
`WaitForEvent()`	Wait until `WFE` instruction completes	*Pseudocode details of the Wait For Event lock mechanism* on page B1-46
`WaitForInterrupt()`	Wait until `WFI` instruction completes	*Pseudocode details of Wait For Interrupt* on page B1-48
`WatchpointDebugEvent()`	Generate a debug event for a watchpoint	*Debug events* on page C3-27

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
`TranslateAddressP()`	アドレス変換を実行し、PMSA メモリアクセスのためのメモリ属性を取得する	*アドレス変換* P. B4-79
`TranslateAddressV()`	アドレス変換を実行し、VMSA メモリアクセスのためのメモリ属性を取得する	
`TranslateVAtoMVA()`	高速コンテキストスイッチ拡張機能の仮想アドレスを修飾仮想アドレスに変換する	*修飾仮想アドレス* P. 付録 E-3
`TranslationTableWalk()`	VMSA 変換テーブルウォークを実行する	*変換テーブルウォーク* P. B3-158
`UInt()`	符号なしの形でビットストリングを整数に変換する	*ビットストリングの整数への変換* P. 付録 I-14
`UnalignedSupport()`	アンアラインド メモリアクセスのサポート（ARMv6 で導入）が使用されているかどうかをチェックする	*UnalignedSupport()* P. 付録 I-26
`UnsignedRecipEstimate()`	符号なし固定小数点の逆数概算	*逆数概算とステップ* P. A2-58
`UnsignedRSqrtEstimate()`	符号なし固定小数点の逆数平方根概算	*逆数平方根* P. A2-61
`UnsignedSat()`	符号なし飽和で整数をビットストリングに変換する	*飽和の擬似コードの詳細* P. A2-9
`UnsignedSatQ()`	符号なし飽和で整数をビットストリングに変換する、飽和フラグの出力あり	
`VCR_OnTakingInterrupt()`	ベクタキャッチの目的で最後に使用された割り込みベクタを追跡する	*ブレークポイントとベクタキャッチ* P. C3-28
`VCRMatch()`	命令ユニットアクセスについてベクキャッチが発生するかどうかをチェックする	
`VCRVectorMatch()`	命令ユニッアクセスがベクタに一致するかどうかをチェックする	
`VectorCatchDebugEvent()`	ベクタキャッチのためのデバッグイベントを生成する	*デバッグイベント* P. C3-27
`VFPExcBarrier()`	すべての VFP 例外処理が発生したことを確認する	*非同期バウンス、直列化、VFP 例外バリア* P. B1-70
`VFPExpandImm()`	VFP 拡張命令におけるイミディエート値の拡張	*動作* P. A7-25

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table J-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`WRPMatch()`	Check whether a data access matches a Watchpoint Register Pair	*Watchpoints* on page C3-35
`ZeroExtend()`	Extend bitstring to left with zero bits	*Zero-extension and sign-extension of bitstrings* on page AppxI-13
`Zeros()`	All-zeros bitstring	*Bitstring concatenation and replication* on page AppxI-12

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 J-2 擬似コードの関数とプロシージャ （続き）

機能	意味	参照先
`VFPSmallRegisterBank()`	16 ダブルワードの VFP 拡張レジスタバンクが実装されている場合は TRUE を返す	*アドバンスト SIMD と VFP の拡張機能レジスタの擬似コードの詳細* P. A2-24
`WaitForEvent()`	`WFE` 命令が完了するまで待つ	*イベント待ちロック機構の擬似コードの詳細* P. B1-46
`WaitForInterrupt()`	`WFI` 命令が完了するまで待つ	*割り込み待ちの擬似コードの詳細* P. B1-48
`WatchpointDebugEvent()`	ウォッチポイントのためのデバッグイベントを生成する	*デバッグイベント* P. C3-27
`WRPMatch()`	データアクセスがウォッチポイント レジスタペアに一致するかどうかをチェックする	*ウォッチポイント* P. C3-35
`ZeroExtend()`	0 のビットでビットストリングを左に延長する	*ビットストリングのゼロ拡張および符号拡張* P. 付録 I-13
`Zeros()`	すべて 0 のビットストリング	*ビットストリングの連結と複製* P. 付録 I-12

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Appendix K
Register Index

This appendix provides an index to the descriptions of the ARM registers in this manual. It contains the following section:

- *Register index* on page AppxK-2.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

付録 K
レジスタ索引

この付録は、本書に記載されている ARM レジスタの説明への索引です。本章は次のセクションから構成されています。

- *レジスタ索引*/ P. 付録 K-2

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

K.1 Register index

Table K-1 shows the main description of each register. The CP15 control coprocessor registers are described separately for VMSA and PMSA implementations, in the sections:

- *CP15 registers for a VMSA implementation* on page B3-64
- *CP15 registers for a PMSA implementation* on page B4-22.

Table K-1 lists both descriptions of these registers. The PMSA and VMSA implementations of a register can differ.

Some CP15 registers are only present in a PMSA implementation, or only in a VMSA implementation. This is shown in Table K-1.

Table K-1 Register index

Register	In[a]	Description, see
Access Permissions, pre-ARMv6		*c5, Memory Region Access Permissions Registers (DAPR and IAPR)* on page AppxH-45
ACTLR	PMSA	*c1, Implementation defined Auxiliary Control Register (ACTLR)* on page B4-50
	VMSA	*c1, Implementation defined Auxiliary Control Register (ACTLR)* on page B3-103
ADFSR	PMSA	*c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B4-56
	VMSA	*c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B3-123
AIDR	PMSA	*c0, Implementation defined Auxiliary ID Register (AIDR)* on page B4-43
	VMSA	*c0, Implementation defined Auxiliary ID Register (AIDR)* on page B3-94
AIFSR	PMSA	*c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B4-56
	VMSA	*c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B3-123
APSR		*The Application Program Status Register (APSR)* on page A2-14
Authentication Status, Debug		*Authentication Status Register (DBGAUTHSTATUS)* on page C10-96
Auxiliary Control	PMSA	*c1, Implementation defined Auxiliary Control Register (ACTLR)* on page B4-50
	VMSA	*c1, Implementation defined Auxiliary Control Register (ACTLR)* on page B3-103

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

K.1 レジスタ索引

表 K-1 は、各レジスタの主な説明を示しています。CP15 の制御コプロセッサレジスタについては、次に示すセクションで VMSA と PMSA の実装に分けて説明されています。

- *VMSA 実装の CP15 レジスタ* P. B3-64
- *PMSA 実装の CP15 レジスタ* P. B4-22

表 K-1 には、これらのレジスタの両方の説明が一覧表記されています。レジスタの PMSA 実装と VMSA 実装は異なる場合があります。

CP15 レジスタの中には、PMSA 実装にのみ存在するもの、または VMSA 実装にのみ存在するものがあります。これらについては、表 K-1 に示されています。

表 K-1 レジスタ索引

レジスタ	実装[a]	説明、参照先
アクセス許可、ARMv6 以前		*c5、メモリ領域アクセス許可レジスタ (DAPR および IAPR)* P. 付録 H-45
ACTLR	PMSA	*c1、実装定義の補助制御レジスタ (ACTLR)* P. B4-50
	VMSA	*c1、実装定義の補助制御レジスタ (ACTLR)* P. B3-103
ADFSR	PMSA	*c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B4-56
	VMSA	*c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B3-123
AIDR	PMSA	*c0、実装定義の補助 ID レジスタ (AIDR)* P. B4-43
	VMSA	*c0、実装定義の補助 ID レジスタ (AIDR)* P. B3-94
AIFSR	PMSA	*c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B4-56
	VMSA	*c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B3-123
APSR		*アプリケーションプログラム ステータスレジスタ (APSR)* P. A2-14
認証ステータス、デバッグ		*認証ステータスレジスタ (DBGAUTHSTATUS)* P. C10-96
補助制御	PMSA	*c1、実装定義の補助制御レジスタ (ACTLR)* P. B4-50
	VMSA	*c1、実装定義の補助制御レジスタ (ACTLR)* P. B3-103

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
Auxiliary Fault Status	PMSA	*c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B4-56
	VMSA	*c5, Auxiliary Data and Instruction Fault Status Registers (ADFSR and AIFSR)* on page B3-123
Auxiliary Feature 0		*c0, Auxiliary Feature Register 0 (ID_AFR0)* on page B5-8
Auxiliary ID	PMSA	*c0, Implementation defined Auxiliary ID Register (AIDR)* on page B4-43
	VMSA	*c0, Implementation defined Auxiliary ID Register (AIDR)* on page B3-94
Block Transfer Status, ARMv6		*c7, Block Transfer Status Register* on page AppxG-43
BPIALL	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
BPIALLIS	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
BPIMVA	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
Breakpoint Control		*Breakpoint Control Registers (DBGBCR)* on page C10-49
Breakpoint value		*Breakpoint Value Registers (DBGBVR)* on page C10-48
c0 - c15	Generic	Instruction descriptions of the `CDP`, `CDP2`, `LDC`, `LDC2`, `MCR`, `MCR2`, `MCRR`, `MCRR2`, `MRC`, `MRC2`, `MRRC`, `MRRC2`, `STC`, and `STC2` generic coprocessor instructions.
	PMSA	*Summary of CP15 register descriptions in a PMSA implementation* on page B4-24
	VMSA	*Summary of CP15 register descriptions in a VMSA implementation* on page B3-66
Cache Behavior Override, ARMv6 Security Extensions		*c9, Cache Behavior Override Register (CBOR)* on page AppxG-49

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
補助フォールトステータス	PMSA	*c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B4-56
	VMSA	*c5、補助データおよび命令フォールト ステータスレジスタ (ADFSR および AIFSR)* P. B3-123
補助機能 0		*c0、補助機能レジスタ 0 (ID_AFR0)* P. B5-8
補助 ID	PMSA	*c0、実装定義の補助 ID レジスタ (AIDR)* P. B4-43
	VMSA	*c0、実装定義の補助 ID レジスタ (AIDR)* P. B3-94
ブロック転送ステータス、ARMv6		*c7、ブロック転送ステータスレジスタ* P. 付録 G-43
BPIALL	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126
BPIALLIS	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126
BPIMVA	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126
ブレークポイント制御		*ブレークポイント制御レジスタ (DBGBCR)* P. C10-49
ブレークポイント値		*ブレークポイント値レジスタ (DBGBVR)* P. C10-48
c0 ～ c15	汎用	`CDP`、`CDP2`、`LDC`、`LDC2`、`MCR`、`MCR2`、`MCRR`、`MCRR2`、`MRC`、`MRC2`、`MRRC`、`MRRC2`、`STC`、`STC2` 汎用コプロセッサ命令の説明
	PMSA	*PMSA 実装の CP15 レジスタの概要* P. B4-24
	VMSA	*VMSA 実装の CP15 レジスタの概要* P. B3-66
キャッシュ動作オーバライド、ARMv6 セキュリティ拡張機能		*c9、キャッシュ動作オーバライドレジスタ (CBOR)* P. 付録 G-49
キャッシュダーティ ステータス、ARMv6		*c7、キャッシュダーティステータス レジスタ (CDSR)* P. 付録 G-39
キャッシュレベル ID	PMSA	*c0、キャッシュレベル ID レジスタ (CLIDR)* P. B4-41
	VMSA	*c0、キャッシュレベル ID レジスタ (CLIDR)* P. B3-92
キャッシュロックダウン、ARMv7 以前		*c9、キャッシュロックダウンのサポート* P. 付録 H-52

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
Cache Dirty Status, ARMv6		*c7, Cache Dirty Status Register (CDSR)* on page AppxG-39
Cache Level ID	PMSA	*c0, Cache Level ID Register (CLIDR)* on page B4-41
	VMSA	*c0, Cache Level ID Register (CLIDR)* on page B3-92
Cache Lockdown, pre-ARMv7		*c9, cache lockdown support* on page AppxH-52
Cache Size ID	PMSA	*c0, Cache Size ID Registers (CCSIDR)* on page B4-40
	VMSA	*c0, Cache Size ID Registers (CCSIDR)* on page B3-91
Cache Type	PMSA	*c0, Cache Type Register (CTR)* on page B4-34
	VMSA	*c0, Cache Type Register (CTR)* on page B3-83
Cacheability, pre-ARMv6		*c2, Memory Region Cacheability Registers (DCR and ICR)* on page AppxH-44
CBOR, ARMv6 Security Extensions		*c9, Cache Behavior Override Register (CBOR)* on page AppxG-49
CCSIDR	PMSA	*c0, Cache Size ID Registers (CCSIDR)* on page B4-40
	VMSA	*c0, Cache Size ID Registers (CCSIDR)* on page B3-91
CDSR, ARMv6		*c7, Cache Dirty Status Register (CDSR)* on page AppxG-39
Claim Tag Clear		*Claim Tag Clear Register (DBGCLAIMCLR)* on page C10-93
Claim Tag Set		*Claim Tag Set Register (DBGCLAIMSET)* on page C10-92
CLIDR	PMSA	*c0, Cache Level ID Register (CLIDR)* on page B4-41
	VMSA	*c0, Cache Level ID Register (CLIDR)* on page B3-92
Component ID		*Debug Component Identification Registers (DBGCID0 to DBGCID3)* on page C10-102
Context ID	PMSA	*c13, Context ID Register (CONTEXTIDR)* on page B4-76
	VMSA	*c13, Context ID Register (CONTEXTIDR)* on page B3-153
Context ID Sampling, Debug		*Context ID Sampling Register (DBGCIDSR)* on page C10-39
CONTEXTIDR	PMSA	*c13, Context ID Register (CONTEXTIDR)* on page B4-76
	VMSA	*c13, Context ID Register (CONTEXTIDR)* on page B3-153

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
キャッシュサイズ ID	PMSA	*c0、キャッシュサイズ ID レジスタ (CCSIDR)* P. B4-40
	VMSA	*c0、キャッシュサイズ ID レジスタ (CCSIDR)* P. B3-91
キャッシュタイプ	PMSA	*c0、キャッシュタイプ レジスタ (CTR)* P. B4-34
	VMSA	*c0、キャッシュタイプ レジスタ (CTR)* P. B3-83
キャッシュ可能、ARMv6 以前		*c2、メモリ領域キャッシュ可能レジスタ （DCR および ICR)* P. 付録 H-44
CBOR、ARMv6 セキュリティ拡張機能		*c9、キャッシュ動作オーバライドレジスタ (CBOR)* P. 付録 G-49
CCSIDR	PMSA	*c0、キャッシュサイズ ID レジスタ (CCSIDR)* P. B4-40
	VMSA	*c0、キャッシュサイズ ID レジスタ (CCSIDR)* P. B3-91
CDSR、ARMv6		*c7、キャッシュダーティステータス レジスタ (CDSR)* P. 付録 G-39
クレームタグ クリア		*クレームタグ クリアレジスタ (DBGCLAIMCLR)* P. C10-93
クレームタグ セット		*クレームタグ セットレジスタ (DBGCLAIMSET)* P. C10-92
CLIDR	PMSA	*c0、キャッシュレベル ID レジスタ (CLIDR)* P. B4-41
	VMSA	*c0、キャッシュレベル ID レジスタ (CLIDR)* P. B3-92
コンポーネント ID		*デバッグコンポーネント識別レジスタ (DBGCID0 ～ DBGCID3)* P. C10-102
コンテキスト ID	PMSA	*c13、コンテキスト ID レジスタ (CONTEXTIDR)* P. B4-76
	VMSA	*c13、コンテキスト ID レジスタ (CONTEXTIDR)* P. B3-153
コンテキスト ID サンプリング、デバッグ		*コンテキスト ID サンプリングレジスタ (DBGCIDSR)* P. C10-39
CONTEXTIDR	PMSA	*c13、コンテキスト ID レジスタ (CONTEXTIDR)* P. B4-76
	VMSA	*c13、コンテキスト ID レジスタ (CONTEXTIDR)* P. B3-153
制御	PMSA	*c1、システム制御レジスタ (SCTLR)* P. B4-45
	VMSA	*c1、システム制御レジスタ (SCTLR)* P. B3-96
コプロセッサアクセス制御	PMSA	*c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B4-51
	VMSA	*c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B3-104

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
Control	PMSA	*c1, System Control Register (SCTLR)* on page B4-45
	VMSA	*c1, System Control Register (SCTLR)* on page B3-96
Coprocessor Access Control	PMSA	*c1, Coprocessor Access Control Register (CPACR)* on page B4-51
	VMSA	*c1, Coprocessor Access Control Register (CPACR)* on page B3-104
Count Enable Clear		*c9, Count Enable Clear Register (PMCNTENCLR)* on page C10-109
Count Enable Set		*c9, Count Enable Set Register (PMCNTENSET)* on page C10-108
CPACR	PMSA	*c1, Coprocessor Access Control Register (CPACR)* on page B4-51
	VMSA	*c1, Coprocessor Access Control Register (CPACR)* on page B3-104
CPSR		*The Current Program Status Register (CPSR)* on page B1-14
CSSELR	PMSA	*c0, Cache Size Selection Register (CSSELR)* on page B4-43
	VMSA	*c0, Cache Size Selection Register (CSSELR)* on page B3-95
CTR	PMSA	*c0, Cache Type Register (CTR)* on page B4-34
	VMSA	*c0, Cache Type Register (CTR)* on page B3-83
Cycle Count		*c9, Cycle Count Register (PMCCNTR)* on page C10-114
D0 - D31		*Advanced SIMD and VFP extension registers* on page A2-21
DACR	VMSA	*c3, Domain Access Control Register (DACR)* on page B3-119
DAPR, pre-ARMv6		*c5, Memory Region Access Permissions Registers (DAPR and IAPR)* on page AppxH-45
Data Fault Address	PMSA	*c6, Data Fault Address Register (DFAR)* on page B4-57
	VMSA	*c6, Data Fault Address Register (DFAR)* on page B3-124
Data Fault Status	PMSA	*c5, Data Fault Status Register (DFSR)* on page B4-55
	VMSA	*c5, Data Fault Status Register (DFSR)* on page B3-121
Data Memory Region Access Permissions, pre-ARMv6		*c5, Memory Region Access Permissions Registers (DAPR and IAPR)* on page AppxH-45
Data Memory Region Bufferability, pre-ARMv6		*c3, Memory Region Bufferability Register (DBR)* on page AppxH-44

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
カウントイネーブル クリア		*c9、カウントイネーブルクリア レジスタ (PMCNTENCLR)* P. C10-109
カウントイネーブル セット		*c9、カウントイネーブルセット レジスタ (PMCNTENSET)* P. C10-108
CPACR	PMSA	*c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B4-51
	VMSA	*c1、コプロセッサアクセス制御レジスタ (CPACR)* P. B3-104
CPSR		*カレントプログラム ステータスレジスタ (CPSR)* P. B1-14
CSSELR	PMSA	*c0、キャッシュサイズ選択レジスタ (CSSELR)* P. B4-43
	VMSA	*c0、キャッシュサイズ選択レジスタ (CSSELR)* P. B3-95
CTR	PMSA	*c0、キャッシュタイプ レジスタ (CTR)* P. B4-34
	VMSA	*c0、キャッシュタイプ レジスタ (CTR)* P. B3-83
サイクルカウント		*c9、サイクルカウント レジスタ (PMCCNTR)* P. C10-114
D0 ～ D31		*アドバンスト SIMD と VFP の拡張機能レジスタ* P. A2-21
DACR	VMSA	*c3、ドメインアクセス制御レジスタ (DACR)* P. B3-119
DAPR、ARMv6 以前		*c5、メモリ領域アクセス許可レジスタ （DAPR および IAPR)* P. 付録 H-45
データフォールト アドレス	PMSA	*c6、データフォールト アドレスレジスタ (DFAR)* P. B4-57
	VMSA	*c6、データフォールト アドレスレジスタ (DFAR)* P. B3-124
データフォールト ステータス	PMSA	*c5、データフォールト ステータスレジスタ (DFSR)* P. B4-55
	VMSA	*c5、データフォールト ステータスレジスタ (DFSR)* P. B3-121
データメモリ領域アクセス許可、ARMv6 以前		*c5、メモリ領域アクセス許可レジスタ （DAPR および IAPR)* P. 付録 H-45
データメモリ領域バッファ可能、ARMv6 以前		*c3、メモリ領域バッファ可能レジスタ (DBR)* P. 付録 H-44
データメモリ領域キャッシュ可能、ARMv6 以前		*c2、メモリ領域キャッシュ可能レジスタ （DCR および ICR)* P. 付録 H-44
データメモリ領域拡張アクセス許可、ARMv6 以前		*c5、メモリ領域拡張アクセス許可レジスタ （DEAPR および IEAPR)* P. 付録 H-46

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
Data Memory Region Cacheability, pre-ARMv6		*c2, Memory Region Cacheability Registers (DCR and ICR)* on page AppxH-44
Data Memory Region Extended Access Permissions, pre-ARMv6		*c5, Memory Region Extended Access Permissions Registers (DEAPR and IEAPR)* on page AppxH-46
Data or unified Cache Lockdown, pre-ARMv7		*c9, cache lockdown support* on page AppxH-52
Data or unified Memory Region, pre-ARMv6		*c6, Memory Region registers (DMRR0-DMRR7 and IMRR0-IMRR7)* on page AppxH-47
Data or unified TLB Lockdown, pre-ARMv7		*c10, VMSA TLB lockdown support* on page AppxH-59
Data Region Access Control	PMSA	*c6, Data Region Access Control Register (DRACR)* on page B4-64
Data Region Base Address	PMSA	*c6, Data Region Base Address Register (DRBAR)* on page B4-60
Data Region Size and Enable	PMSA	*c6, Data Region Size and Enable Register (DRSR)* on page B4-62
Data TCM Non-Secure Access Control, ARMv6		*c9, TCM Non-Secure Access Control Registers, DTCM-NSACR and ITCM-NSACR* on page AppxG-51
Data TCM Region, ARMv6		*c9, TCM Region Registers (DTCMRR and ITCMRR)* on page AppxG-47
Data Transfer, Debug		*Host to Target Data Transfer Register (DBGDTRRX)* on page C10-40 *Target to Host Data Transfer Register (DBGDTRTX)* on page C10-43
DBGAUTHSTATUS		*Authentication Status Register (DBGAUTHSTATUS)* on page C10-96
DBGBCR0 - DBGBCR15		*Breakpoint Control Registers (DBGBCR)* on page C10-49
DBGBVR0 - DBGBVR15		*Breakpoint Value Registers (DBGBVR)* on page C10-48
DBGCID0 - DBGCID3		*Debug Component Identification Registers (DBGCID0 to DBGCID3)* on page C10-102
DBGCIDSR		*Context ID Sampling Register (DBGCIDSR)* on page C10-39
DBGCLAIMCLR		*Claim Tag Clear Register (DBGCLAIMCLR)* on page C10-93
DBGCLAIMSET		*Claim Tag Set Register (DBGCLAIMSET)* on page C10-92
DBGDEVID		*Debug Device ID Register (DBGDEVID)* on page C10-6.
DBGDEVTYPE		*Device Type Register (DBGDEVTYPE)* on page C10-98

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
データまたは統一キャッシュロックダウン、ARMv7 以前		*c9、キャッシュロックダウンのサポート* P. 付録 H-52
データまたは統一メモリ領域、ARMv6 以前		*c6、メモリ領域レジスタ （DMRR0 ～ DMRR7 および IMRR0 ～ IMRR7）* P. 付録 H-47
データまたは統一 TLB ロックダウン、ARMv7 以前		*c10、VMSA TLB ロックダウンのサポート* P. 付録 H-59
データ領域アクセス制御	PMSA	*c6、データ領域アクセス制御レジスタ (DRACR)* P. B4-64
データ領域ベースアドレス	PMSA	*c6、データ領域ベースアドレス レジスタ (DRBAR)* P. B4-60
データ領域サイズ / イネーブル	PMSA	*c6、データ領域サイズおよびイネーブルレジスタ (DRSR)* P. B4-62
データ TCM 非セキュアアクセス制御、ARMv6		*c9、TCM 非セキュアアクセス制御レジスタ、DTCM-NSACR および ITCM-NSACR* P. 付録 G-51
データ TCM 領域、ARMv6		*c9、TCM 領域レジスタ (DTCMRR および ITCMRR)* P. 付録 G-47
データ転送、デバッグ		*ホストからターゲットへのデータ転送レジスタ (DBGDTRRX)* P. C10-40 *ターゲットからホストへのデータ転送レジスタ (DBGDTRTX)* P. C10-43
DBGAUTHSTATUS		*認証ステータスレジスタ (DBGAUTHSTATUS)* P. C10-96
DBGBCR0 ～ DBGBCR15		*ブレークポイント制御レジスタ (DBGBCR)* P. C10-49
DBGBVR0 ～ DBGBVR15		*ブレークポイント値レジスタ (DBGBVR)* P. C10-48
DBGCID0 ～ DBGCID3		*デバッグコンポーネント識別レジスタ (DBGCID0 ～ DBGCID3)* P. C10-102
DBGCIDSR		*コンテキスト ID サンプリングレジスタ (DBGCIDSR)* P. C10-39
DBGCLAIMCLR		*クレームタグ クリアレジスタ (DBGCLAIMCLR)* P. C10-93
DBGCLAIMSET		*クレームタグ セットレジスタ (DBGCLAIMSET)* P. C10-92
DBGDEVID		*デバッグデバイス ID レジスタ (DBGDEVID)* P. C10-6
DBGDEVTYPE		*デバイスタイプ レジスタ (DBGDEVTYPE)* P. C10-98
DBGDIDR		*デバッグ ID レジスタ (DBGDIDR)* P. C10-3
DBGDRAR		*デバッグ ROM アドレスレジスタ (DBGDRAR)* P. C10-7

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
DBGDIDR		*Debug ID Register (DBGDIDR)* on page C10-3
DBGDRAR		*Debug ROM Address Register (DBGDRAR)* on page C10-7
DBGDRCR		*Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29
DBGDSAR		*Debug Self Address Offset Register (DBGDSAR)* on page C10-8
DBGDSCCR		*Debug State Cache Control Register (DBGDSCCR)* on page C10-81
DBGDSCR		*Debug Status and Control Register (DBGDSCR)* on page C10-10
DBGDSCRext		*Internal and external views of the DBGDSCR and the DCC registers* on page C6-21
DBGDSCRint		*Internal and external views of the DBGDSCR and the DCC registers* on page C6-21
DBGDSMCR		*Debug State MMU Control Register (DBGDSMCR)* on page C10-84
DBGDTRRX		*Host to Target Data Transfer Register (DBGDTRRX)* on page C10-40
DBGDTRRXext		*Internal and external views of the DBGDSCR and the DCC registers* on page C6-21
DBGDTRRXint		*Internal and external views of the DBGDSCR and the DCC registers* on page C6-21
DBGDTRTX		*Target to Host Data Transfer Register (DBGDTRTX)* on page C10-43
DBGDTRTXext		*Internal and external views of the DBGDSCR and the DCC registers* on page C6-21
DBGDTRTXint		*Internal and external views of the DBGDSCR and the DCC registers* on page C6-21
DBGECR		*Event Catch Register (DBGECR)* on page C10-78
DBGITCTRL		*Integration Mode Control Register (DBGITCTRL)* on page C10-91
DBGITR		*Instruction Transfer Register (DBGITR)* on page C10-46
DBGLAR		*Lock Access Register (DBGLAR)* on page C10-94
DBGLSR		*Lock Status Register (DBGLSR)* on page C10-95
DBGOSLAR		*OS Lock Access Register (DBGOSLAR)* on page C10-75

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
DBGDRCR		*デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10-29
DBGDSAR		*デバッグ自己アドレスオフセット レジスタ (DBGDSAR)* P. C10-8
DBGDSCCR		*デバッグ状態キャッシュ制御レジスタ (DBGDSCCR)* P. C10-81
DBGDSCR		*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10
DBGDSCRext		*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6-21
DBGDSCRint		*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6-21
DBGDSMCR		*デバッグ状態 MMU 制御レジスタ (DBGDSMCR)* P. C10-84
DBGDTRRX		*ホストからターゲットへのデータ転送レジスタ (DBGDTRRX)* P. C10-40
DBGDTRRXext		*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6-21
DBGDTRRXint		*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6-21
DBGDTRTX		*ターゲットからホストへのデータ転送レジスタ (DBGDTRTX)* P. C10-43
DBGDTRTXext		*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6-21
DBGDTRTXint		*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6-21
DBGECR		*イベントキャッチ レジスタ (DBGECR)* P. C10-78
DBGITCTRL		*統合モード制御レジスタ (DBGITCTRL)* P. C10-91
DBGITR		*命令転送レジスタ (DBGITR)* P. C10-46
DBGLAR		*ロックアクセス レジスタ (DBGLAR)* P. C10-94
DBGLSR		*ロックステータス レジスタ (DBGLSR)* P. C10-95
DBGOSLAR		*OS ロックアクセス レジスタ (DBGOSLAR)* P. C10-75
DBGOSLSR		*OS ロックステータス レジスタ (DBGOSLSR)* P. C10-76

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
DBGOSLSR		*OS Lock Status Register (DBGOSLSR)* on page C10-76
DBGOSSRR		*OS Save and Restore Register (DBGOSSRR)* on page C10-77
DBGPCSR		*Program Counter Sampling Register (DBGPCSR)* on page C10-38
DBGPID0 - DBGPID4		*Debug Peripheral Identification Registers (DBGPID0 to DBGPID4)* on page C10-98
DBGPRCR		*Device Power-down and Reset Control Register (DBGPRCR), v7 Debug only* on page C10-31
DBGPRSR		*Device Power-down and Reset Status Register (DBGPRSR), v7 Debug only* on page C10-34
DBGVCR		*Vector Catch Register (DBGVCR)* on page C10-67
DBGWCR0 - DBGWCR15		*Watchpoint Control Registers (DBGWCR)* on page C10-61
DBGWFAR, CP14		*Watchpoint Fault Address Register (DBGWFAR)* on page C10-28
DBGWFAR, CP15, ARMv6		*c6, Watchpoint Fault Address Register (DBGWFAR)* on page AppxG-37
DBGWVR0 - DBGWVR15		*Watchpoint Value Registers (DBGWVR)* on page C10-60
DBR, pre-ARMv6		*c3, Memory Region Bufferability Register (DBR)* on page AppxH-44
DCC		*Internal and external views of the DBGDSCR and the DCC registers* on page C6-21
DCCIMVAC	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
DCCISW	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
DCCMVAC	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
DBGOSSRR		*OS 保存および復元レジスタ (DBGOSSRR)* P. C10-77
DBGPCSR		*プログラムカウンタサンプリング レジスタ (DBGPCSR)* P. C10-38
DBGPID0 ～ DBGPID4		*デバッグペリフェラル識別レジスタ (DBGPID0 ～ DBGPID4)* P. C10-98
DBGPRCR		*デバイス電力オフおよびリセット制御レジスタ (DBGPRCR)、v7 デバッグのみ* P. C10-31
DBGPRSR		*デバイス電力オフおよびリセット ステータス レジスタ (DBGPRSR)、v7 デバッグのみ* P. C10-34
DBGVCR		*ベクタキャッチ レジスタ (DBGVCR)* P. C10-67
DBGWCR0 ～ DBGWCR15		*ウォッチポイント制御レジスタ (DBGWCR)* P. C10-61
DBGWFAR、CP14		*ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10-28
DBGWFAR、CP15、ARMv6		*c6、ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. 付録 G-37
DBGWVR0 ～ DBGWVR15		*ウォッチポイント値レジスタ (DBGWVR)* P. C10-60
DBR、ARMv6 以前		*c3、メモリ領域バッファ可能レジスタ (DBR)* P. 付録 H-44
DCC		*DBGDSCR および DCC レジスタの内部および外部ビュー* P. C6-21
DCCIMVAC	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126
DCCISW	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126
DCCMVAC	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126
DCCMVAU	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
DCCMVAU	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
DCCSW	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
DCIMVAC	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
DCISW	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
DCLR, pre-ARMv7		*c9, cache lockdown support* on page AppxH-52
DCLR2, pre-ARMv7		*c9, Format D Data or unified Cache Lockdown Register, DCLR2* on page AppxH-58
DCR, pre-ARMv6		*c2, Memory Region Cacheability Registers (DCR and ICR)* on page AppxH-44
DEAPR, pre-ARMv6		*c5, Memory Region Extended Access Permissions Registers (DEAPR and IEAPR)* on page AppxH-46
Debug Component ID		*Debug Component Identification Registers (DBGCID0 to DBGCID3)* on page C10-102
Debug Context ID Sampling		*Context ID Sampling Register (DBGCIDSR)* on page C10-39
Debug Device ID		*Debug Device ID Register (DBGDEVID)* on page C10-6.
Debug Feature 0		*c0, Debug Feature Register 0 (ID_DFR0)* on page B5-6
Debug ID		*Debug ID Register (DBGDIDR)* on page C10-3

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
DCCSW	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126
DCIMVAC	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126
DCISW	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126
DCLR、ARMv7 以前		*c9、キャッシュロックダウンのサポート* P. 付録 H-52
DCLR2、ARMv7 以前		*c9、フォーマット D のデータまたは統一キャッシュロックダウン レジスタ、DCLR2* P. 付録 H-58
DCR、ARMv6 以前		*c2、メモリ領域キャッシュ可能レジスタ （DCR および ICR）* P. 付録 H-44
DEAPR、ARMv6 以前		*c5、メモリ領域拡張アクセス許可レジスタ （DEAPR および IEAPR）* P. 付録 H-46
デバッグコンポーネント ID		*デバッグコンポーネント識別レジスタ (DBGCID0 ～ DBGCID3)* P. C10-102
デバッグコンテキスト ID サンプリング		*コンテキスト ID サンプリングレジスタ (DBGCIDSR)* P. C10-39
デバッグデバイス ID		*デバッグデバイス ID レジスタ (DBGDEVID)* P. C10-6
デバッグ機能 0		*c0、デバッグ機能レジスタ 0 (ID_DFR0)* P. B5-6
デバッグ ID		*デバッグ ID レジスタ (DBGDIDR)* P. C10-3
デバッグペリフェラル ID		*デバッグペリフェラル識別レジスタ (DBGPID0 ～ DBGPID4)* P. C10-98
デバッグ プログラムカウンタ サンプリング		*プログラムカウンタサンプリング レジスタ (DBGPCSR)* P. C10-38
デバッグ ROM アドレス		*デバッグ ROM アドレスレジスタ (DBGDRAR)* P. C10-7
デバッグ実行制御		*デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10-29
デバッグ自己アドレスオフセット		*デバッグ自己アドレスオフセット レジスタ (DBGDSAR)* P. C10-8
デバッグ状態キャッシュ制御		*デバッグ状態キャッシュ制御レジスタ (DBGDSCCR)* P. C10-81

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
Debug Peripheral ID		*Debug Peripheral Identification Registers (DBGPID0 to DBGPID4)* on page C10-98
Debug Program Counter Sampling		*Program Counter Sampling Register (DBGPCSR)* on page C10-38
Debug ROM Address		*Debug ROM Address Register (DBGDRAR)* on page C10-7
Debug Run Control		*Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29
Debug Self Address Offset		*Debug Self Address Offset Register (DBGDSAR)* on page C10-8
Debug State Cache Control		*Debug State Cache Control Register (DBGDSCCR)* on page C10-81
Debug State MMU Control		*Debug State MMU Control Register (DBGDSMCR)* on page C10-84
Debug Status and Control		*Debug Status and Control Register (DBGDSCR)* on page C10-10
Device ID, Debug		*Debug Device ID Register (DBGDEVID)* on page C10-6
Device Power-down and Reset Control		*Device Power-down and Reset Control Register (DBGPRCR), v7 Debug only* on page C10-31
Device Power-down and Reset Status		*Device Power-down and Reset Status Register (DBGPRSR), v7 Debug only* on page C10-34
Device Type, Debug		*Device Type Register (DBGDEVTYPE)* on page C10-98
DFAR	PMSA	*c6, Data Fault Address Register (DFAR)* on page B4-57
	VMSA	*c6, Data Fault Address Register (DFAR)* on page B3-124
DFSR	PMSA	*c5, Data Fault Status Register (DFSR)* on page B4-55
	VMSA	*c5, Data Fault Status Register (DFSR)* on page B3-121
DMRR0-DMRR7, pre-ARMv6		*c6, Memory Region registers (DMRR0-DMRR7 and IMRR0-IMRR7)* on page AppxH-47
Domain Access Control	VMSA	*c3, Domain Access Control Register (DACR)* on page B3-119
DRACR	PMSA	*c6, Data Region Access Control Register (DRACR)* on page B4-64
DRBAR	PMSA	*c6, Data Region Base Address Register (DRBAR)* on page B4-60
DRSR	PMSA	*c6, Data Region Size and Enable Register (DRSR)* on page B4-62
DTCM-NSACR, ARMv6		*c9, TCM Non-Secure Access Control Registers, DTCM-NSACR and ITCM-NSACR* on page AppxG-51

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
デバッグ状態 MMU 制御		*デバッグ状態 MMU 制御レジスタ (DBGDSMCR)* P. C10-84
デバッグステータスおよび制御		*デバッグステータスおよび制御レジスタ (DBGDSCR)* P. C10-10
デバイス ID、デバッグ		*デバッグデバイス ID レジスタ (DBGDEVID)* P. C10-6
デバイス電力オフおよびリセット制御		*デバイス電力オフおよびリセット制御レジスタ (DBGPRCR)、v7 デバッグのみ* P. C10-31
デバイス電力オフおよびリセットステータス		*デバイス電力オフおよびリセット ステータス レジスタ (DBGPRSR)、v7 デバッグのみ* P. C10-34
デバイスタイプ、デバッグ		*デバイスタイプ レジスタ (DBGDEVTYPE)* P. C10-98
DFAR	PMSA	*c6、データフォールト アドレスレジスタ (DFAR)* P. B4-57
	VMSA	*c6、データフォールト アドレスレジスタ (DFAR)* P. B3-124
DFSR	PMSA	*c5、データフォールト ステータスレジスタ (DFSR)* P. B4-55
	VMSA	*c5、データフォールト ステータスレジスタ (DFSR)* P. B3-121
DMRR0 ～ DMRR7、ARMv6 以前		*c6、メモリ領域レジスタ （DMRR0 ～ DMRR7 および IMRR0 ～ IMRR7)* P. 付録 H-47
ドメインアクセス制御	VMSA	*c3、ドメインアクセス制御レジスタ (DACR)* P. B3-119
DRACR	PMSA	*c6、データ領域アクセス制御レジスタ (DRACR)* P. B4-64
DRBAR	PMSA	*c6、データ領域ベースアドレス レジスタ (DRBAR)* P. B4-60
DRSR	PMSA	*c6、データ領域サイズおよびイネーブルレジスタ (DRSR)* P. B4-62
DTCM ～ NSACR、ARMv6		*c9、TCM 非セキュアアクセス制御レジスタ、DTCM-NSACR および ITCM-NSACR* P. 付録 G-51
DTCMRR、ARMv6		*c9、TCM 領域レジスタ (DTCMRR および ITCMRR)* P. 付録 G-47
DTLBIALL	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
DTLBIASID	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
DTLBIMVA	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
DTLBLR、ARMv7 以前		*c10、VMSA TLB ロックダウンのサポート* P. 付録 H-59
ENDIANSTATE		*ENDIANSTATE* P. A2-19

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
DTCMRR, ARMv6		*c9, TCM Region Registers (DTCMRR and ITCMRR)* on page AppxG-47
DTLBIALL	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
DTLBIASID	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
DTLBIMVA	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
DTLBLR, pre-ARMv7		*c10, VMSA TLB lockdown support* on page AppxH-59
ENDIANSTATE		*ENDIANSTATE* on page A2-19
Event		*The Event Register* on page B1-46
Event Catch		*Event Catch Register (DBGECR)* on page C10-78
Event Count		*c9, Event Count Register (PMXEVCNTR)* on page C10-116
Event Counter Selection		*c9, Event Counter Selection Register (PMSELR)* on page C10-113
Event Select		*c9, Event Type Select Register (PMXEVTYPER)* on page C10-115
Extended Access Permissions, pre-ARMv6		*c5, Memory Region Extended Access Permissions Registers (DEAPR and IEAPR)* on page AppxH-46
FAR		See *Fault Address*
Fault Address		*c6, Data Fault Address Register (DFAR)* on page B4-57 (PMSA) *c6, Data Fault Address Register (DFAR)* on page B3-124 (VMSA) *c6, Instruction Fault Address Register (IFAR)* on page B4-58 (PMSA) *c6, Instruction Fault Address Register (IFAR)* on page B3-125 (VMSA) *Watchpoint Fault Address Register (DBGWFAR)* on page C10-28 *c6, Watchpoint Fault Address Register (DBGWFAR)* on page AppxG-37 (ARMv6)
Fault Status	PMSA	*c5, Data Fault Status Register (DFSR)* on page B4-55 *c5, Instruction Fault Status Register (IFSR)* on page B4-56
	VMSA	*c5, Data Fault Status Register (DFSR)* on page B3-121 *c5, Instruction Fault Status Register (IFSR)* on page B3-122
FCSE Process ID	VMSA	*c13, FCSE Process ID Register (FCSEIDR)* on page B3-152
FCSEIDR	VMSA	*c13, FCSE Process ID Register (FCSEIDR)* on page B3-152
Floating-point Exception		*The Floating-Point Exception Register (FPEXC)* on page B1-68

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
イベント		*イベントレジスタ* P. B1-46
イベントキャッチ		*イベントキャッチ レジスタ (DBGECR)* P. C10-78
イベントカウント		*c9、イベントカウント レジスタ (PMXEVCNTR)* P. C10-116
イベントカウンタ選択		*c9、イベントカウンタ選択レジスタ (PMSELR)* P. C10-113
イベント選択		*c9、イベントタイプ選択レジスタ (PMXEVTYPER)* P. C10-115
拡張アクセス許可、ARMv6 以前		*c5、メモリ領域拡張アクセス許可レジスタ （DEAPR および IEAPR)* P. 付録 H-46
FAR		*フォールトアドレス*を参照
フォールトアドレス		*c6、データフォールト アドレスレジスタ (DFAR)* P. B4-57 (PMSA) *c6、データフォールト アドレスレジスタ (DFAR)* P. B3-124 (VMSA) *c6、命令フォールト アドレスレジスタ (IFAR)* P. B4-58 (PMSA) *c6、命令フォールト アドレスレジスタ (IFAR)* P. B3-125 (VMSA) *ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10-28 *c6、ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. 付録 G-37 (ARMv6)
フォールトステータス	PMSA	*c5、データフォールト ステータスレジスタ (DFSR)* P. B4-55 *c5、命令フォールト ステータスレジスタ (IFSR)* P. B4-56
	VMSA	*c5、データフォールト ステータスレジスタ (DFSR)* P. B3-121 *c5、命令フォールト ステータスレジスタ (IFSR)* P. B3-122
FCSE プロセス ID	VMSA	*c13、FCSE プロセス ID レジスタ (FCSEIDR)* P. B3-152
FCSEIDR	VMSA	*c13、FCSE プロセス ID レジスタ (FCSEIDR)* P. B3-152
浮動小数点例外		*浮動小数点例外レジスタ (FPEXC)* P. B1-68
浮動小数点命令		*浮動小数点命令レジスタ、FPINST および FPINST2* P. 付録 B-20
浮動小数点システム ID		*浮動小数点システム ID レジスタ (FPSID)* P. B5-34
フォーマット D のデータキャッシュロックダウン、ARMv7 以前		*c9、フォーマット D のデータまたは統一キャッシュロックダウン レジスタ、DCLR2* P. 付録 H-58
FPEXC		*浮動小数点例外レジスタ (FPEXC)* P. B1-68

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
Floating-point Instruction		*The Floating-Point Instruction Registers, FPINST and FPINST2* on page AppxB-20
Floating-point System ID		*Floating-point System ID Register (FPSID)* on page B5-34
Format D Data Cache Lockdown, pre-ARMv7		*c9, Format D Data or unified Cache Lockdown Register, DCLR2* on page AppxH-58
FPEXC		*The Floating-Point Exception Register (FPEXC)* on page B1-68
FPINST		*The Floating-Point Instruction Registers, FPINST and FPINST2* on page AppxB-20
FPINST2		*The Floating-Point Instruction Registers, FPINST and FPINST2* on page AppxB-20
FPSCR		*Floating-point Status and Control Register (FPSCR)* on page A2-28
FPSID		*Floating-point System ID Register (FPSID)* on page B5-34
FSR		See *Fault Status*
Host to Target Data Transfer		*Host to Target Data Transfer Register (DBGDTRRX)* on page C10-40
IAPR, pre-ARMv6		*c5, Memory Region Access Permissions Registers (DAPR and IAPR)* on page AppxH-45
ICIALLU	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
ICIALLUIS	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
ICIMVAU	PMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B4-68
	VMSA	*CP15 c7, Cache and branch predictor maintenance functions* on page B3-126
ICLR, pre-ARMv7		*c9, cache lockdown support* on page AppxH-52

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
FPINST		*浮動小数点命令レジスタ、FPINST および FPINST2* P. 付録 B-20
FPINST2		*浮動小数点命令レジスタ、FPINST および FPINST2* P. 付録 B-20
FPSCR		*浮動小数点ステータスおよび制御レジスタ (FPSCR)* P. A2-28
FPSID		*浮動小数点システム ID レジスタ (FPSID)* P. B5-34
FSR		*フォールトステータスを参照*
ホストからターゲットへのデータ転送		*ホストからターゲットへのデータ転送レジスタ (DBGDTRRX)* P. C10-40
IAPR、ARMv6 以前		*c5、メモリ領域アクセス許可レジスタ （DAPR および IAPR)* P. 付録 H-45
ICIALLU	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126
ICIALLUIS	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126
ICIMVAU	PMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B4-68
	VMSA	*CP15 c7、キャッシュおよび分岐予測器保守機能* P. B3-126
ICLR、ARMv7 以前		*c9、キャッシュロックダウンのサポート* P. 付録 H-52
ICR、ARMv6 以前		*c2、メモリ領域キャッシュ可能レジスタ （DCR および ICR)* P. 付録 H-44
ID_AFR0		*c0、補助機能レジスタ 0 (ID_AFR0)* P. B5-8
ID_DFR0		*c0、デバッグ機能レジスタ 0 (ID_DFR0)* P. B5-6
ID_ISAR0		*c0、命令セット属性レジスタ 0 (ID_ISAR0)* P. B5-24
ID_ISAR1		*c0、命令セット属性レジスタ 1 (ID_ISAR1)* P. B5-25
ID_ISAR2		*c0、命令セット属性レジスタ 2 (ID_ISAR2)* P. B5-27
ID_ISAR3		*c0、命令セット属性レジスタ 3 (ID_ISAR3)* P. B5-29
ID_ISAR4		*c0、命令セット属性レジスタ 4 (ID_ISAR4)* P. B5-31
ID_ISAR5		*c0、命令セット属性レジスタ 5 (ID_ISAR5)* P. B5-33

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
ICR, pre-ARMv6		*c2, Memory Region Cacheability Registers (DCR and ICR)* on page AppxH-44
ID_AFR0		*c0, Auxiliary Feature Register 0 (ID_AFR0)* on page B5-8
ID_DFR0		*c0, Debug Feature Register 0 (ID_DFR0)* on page B5-6
ID_ISAR0		*c0, Instruction Set Attribute Register 0 (ID_ISAR0)* on page B5-24
ID_ISAR1		*c0, Instruction Set Attribute Register 1 (ID_ISAR1)* on page B5-25
ID_ISAR2		*c0, Instruction Set Attribute Register 2 (ID_ISAR2)* on page B5-27
ID_ISAR3		*c0, Instruction Set Attribute Register 3 (ID_ISAR3)* on page B5-29
ID_ISAR4		*c0, Instruction Set Attribute Register 4 (ID_ISAR4)* on page B5-31
ID_ISAR5		*c0, Instruction Set Attribute Register 5 (ID_ISAR5)* on page B5-33
ID_MMFR0		*c0, Memory Model Feature Register 0 (ID_MMFR0)* on page B5-9
ID_MMFR1		*c0, Memory Model Feature Register 1 (ID_MMFR1)* on page B5-11
ID_MMFR2		*c0, Memory Model Feature Register 2 (ID_MMFR2)* on page B5-14
ID_MMFR3		*c0, Memory Model Feature Register 3 (ID_MMFR3)* on page B5-17
ID_PFR0		*c0, Processor Feature Register 0 (ID_PFR0)* on page B5-4
ID_PFR1		*c0, Processor Feature Register 1 (ID_PFR1)* on page B5-5
ID, Debug		*Debug ID Register (DBGDIDR)* on page C10-3
IEAPR, pre-ARMv6		*c5, Memory Region Extended Access Permissions Registers (DEAPR and IEAPR)* on page AppxH-46
IFAR	PMSA	*c6, Instruction Fault Address Register (IFAR)* on page B4-58
	VMSA	*c6, Instruction Fault Address Register (IFAR)* on page B3-125
IFSR	PMSA	*c5, Instruction Fault Status Register (IFSR)* on page B4-56
	VMSA	*c5, Instruction Fault Status Register (IFSR)* on page B3-122
IMRR0-IMRR7, pre-ARMv6		*c6, Memory Region registers (DMRR0-DMRR7 and IMRR0-IMRR7)* on page AppxH-47
Instruction Cache Lockdown, pre-ARMv7		*c9, cache lockdown support* on page AppxH-52

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
ID_MMFR0		*c0、メモリモデル機能レジスタ 0 (ID_MMFR0)* P. B5-9
ID_MMFR1		*c0、メモリモデル機能レジスタ 1 (ID_MMFR1)* P. B5-11
ID_MMFR2		*c0、メモリモデル機能レジスタ 2 (ID_MMFR2)* P. B5-14
ID_MMFR3		*c0、メモリモデル機能レジスタ 3 (ID_MMFR3)* P. B5-17
ID_PFR0		*c0、プロセッサ機能レジスタ 0 (ID_PFR0)* P. B5-4
ID_PFR1		*c0、プロセッサ機能レジスタ 1 (ID_PFR1)* P. B5-5
ID、デバッグ		*デバッグ ID レジスタ (DBGDIDR)* P. C10-3
IEAPR、ARMv6 以前		*c5、メモリ領域拡張アクセス許可レジスタ （DEAPR および IEAPR）* P. 付録 H-46
IFAR	PMSA	*c6、命令フォールト アドレスレジスタ (IFAR)* P. B4-58
	VMSA	*c6、命令フォールト アドレスレジスタ (IFAR)* P. B3-125
IFSR	PMSA	*c5、命令フォールト ステータスレジスタ (IFSR)* P. B4-56
	VMSA	*c5、命令フォールト ステータスレジスタ (IFSR)* P. B3-122
IMRR0 ～ IMRR7、ARMv6 以前		*c6、メモリ領域レジスタ （DMRR0 ～ DMRR7 および IMRR0 ～ IMRR7）* P. 付録 H-47
命令キャッシュロックダウン、ARMv7 以前		*c9、キャッシュロックダウンのサポート* P. 付録 H-52
命令フォールトアドレス	PMSA	*c6、命令フォールト アドレスレジスタ (IFAR)* P. B4-58
	VMSA	*c6、命令フォールト アドレスレジスタ (IFAR)* P. B3-125
命令フォールトステータス	PMSA	*c5、命令フォールト ステータスレジスタ (IFSR)* P. B4-56
	VMSA	*c5、命令フォールト ステータスレジスタ (IFSR)* P. B3-122
命令メモリ領域キャッシュ可能、ARMv6 以前		*c2、メモリ領域キャッシュ可能レジスタ （DCR および ICR）* P. 付録 H-44
命令メモリ領域拡張アクセス許可、ARMv6 以前		*c5、メモリ領域拡張アクセス許可レジスタ （DEAPR および IEAPR）* P. 付録 H-46
命令メモリ領域、ARMv6 以前		*c6、メモリ領域レジスタ （DMRR0 ～ DMRR7 および IMRR0 ～ IMRR7）* P. 付録 H-47

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
Instruction Fault Address	PMSA	*c6, Instruction Fault Address Register (IFAR)* on page B4-58
	VMSA	*c6, Instruction Fault Address Register (IFAR)* on page B3-125
Instruction Fault Status	PMSA	*c5, Instruction Fault Status Register (IFSR)* on page B4-56
	VMSA	*c5, Instruction Fault Status Register (IFSR)* on page B3-122
Instruction Memory Region Cacheability, pre-ARMv6		*c2, Memory Region Cacheability Registers (DCR and ICR)* on page AppxH-44
Instruction Memory Region Extended Access Permissions, pre-ARMv6		*c5, Memory Region Extended Access Permissions Registers (DEAPR and IEAPR)* on page AppxH-46
Instruction Memory Region, pre-ARMv6		*c6, Memory Region registers (DMRR0-DMRR7 and IMRR0-IMRR7)* on page AppxH-47
Instruction Memory Region Access Permissions, pre-ARMv6		*c5, Memory Region Access Permissions Registers (DAPR and IAPR)* on page AppxH-45
Instruction Region Access Control	PMSA	*c6, Instruction Region Access Control Register (IRACR)* on page B4-65
Instruction Region Base Address	PMSA	*c6, Instruction Region Base Address Register (IRBAR)* on page B4-61
Instruction Region Size and Enable	PMSA	*c6, Instruction Region Size and Enable Register (IRSR)* on page B4-63
Instruction Set Attribute		*CP15 c0, Instruction Set Attribute registers* on page B5-19
Instruction TCM Non-Secure Access Control, ARMv6		*c9, TCM Non-Secure Access Control Registers, DTCM-NSACR and ITCM-NSACR* on page AppxG-51
Instruction TCM Region, ARMv6		*c9, TCM Region Registers (DTCMRR and ITCMRR)* on page AppxG-47
Instruction TLB Lockdown Register, pre-ARMv7		*c10, VMSA TLB lockdown support* on page AppxH-59
Instruction Transfer Register, Debug		*Instruction Transfer Register (DBGITR)* on page C10-46
Integration Mode Control		*Integration Mode Control Register (DBGITCTRL)* on page C10-91
Interrupt Enable Clear		*c9, Interrupt Enable Clear Register (PMINTENCLR)* on page C10-119
Interrupt Enable Set		*c9, Interrupt Enable Set Register (PMINTENSET)* on page C10-118
Interrupt Status	VMSA	*c12, Interrupt Status Register (ISR)* on page B3-150

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
命令メモリ領域アクセス許可、ARMv6 以前		*c5、メモリ領域アクセス許可レジスタ (DAPR および IAPR)* P. 付録 H-45
命令領域アクセス制御	PMSA	*c6、命令領域アクセス制御レジスタ (IRACR)* P. B4-65
命令領域ベースアドレス	PMSA	*c6、命令領域ベースアドレス レジスタ (IRBAR)* P. B4-61
命令領域サイズおよびイネーブル	PMSA	*c6、命令領域サイズおよびイネーブルレジスタ (IRSR)* P. B4-63
命令セット属性		*CP15 c0、命令セット属性レジスタ* P. B5-19
命令 TCM 非セキュアアクセス制御、ARMv6		*c9、TCM 非セキュアアクセス制御レジスタ、DTCM-NSACR および ITCM-NSACR* P. 付録 G-51
命令 TCM 領域、ARMv6		*c9、TCM 領域レジスタ (DTCMRR および ITCMRR)* P. 付録 G-47
命令 TLB ロックダウンレジスタ、ARMv7 以前		*c10、VMSA TLB ロックダウンのサポート* P. 付録 H-59
命令転送レジスタ、デバッグ		*命令転送レジスタ (DBGITR)* P. C10-46
統合モード制御		*統合モード制御レジスタ (DBGITCTRL)* P. C10-91
割り込みイネーブルクリア		*c9、割り込みイネーブルクリア レジスタ (PMINTENCLR)* P. C10-119
割り込みイネーブルセット		*c9、割り込みイネーブルセット レジスタ (PMINTENSET)* P. C10-118
割り込みステータス	VMSA	*c12、割り込みステータスレジスタ (ISR)* P. B3-150
IRACR	PMSA	*c6、命令領域アクセス制御レジスタ (IRACR)* P. B4-65
IRBAR	PMSA	*c6、命令領域ベースアドレス レジスタ (IRBAR)* P. B4-61
IRSR	PMSA	*c6、命令領域サイズおよびイネーブルレジスタ (IRSR)* P. B4-63
ISETSTATE		*ISETSTATE* P. A2-15
ISR	VMSA	*c12、割り込みステータスレジスタ (ISR)* P. B3-150
ITCM-NSACR、ARMv6		*c9、TCM 非セキュアアクセス制御レジスタ、DTCM-NSACR および ITCM-NSACR* P. 付録 G-51
ITCMRR、ARMv6		*c9、TCM 領域レジスタ (DTCMRR および ITCMRR)* P. 付録 G-47
ITLBIALL	VMSA	*CP15 c8、TLB 保守操作* P. B3-138

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
IRACR	PMSA	*c6, Instruction Region Access Control Register (IRACR)* on page B4-65
IRBAR	PMSA	*c6, Instruction Region Base Address Register (IRBAR)* on page B4-61
IRSR	PMSA	*c6, Instruction Region Size and Enable Register (IRSR)* on page B4-63
ISETSTATE		*ISETSTATE* on page A2-15
ISR	VMSA	*c12, Interrupt Status Register (ISR)* on page B3-150
ITCM-NSACR, ARMv6		*c9, TCM Non-Secure Access Control Registers, DTCM-NSACR and ITCM-NSACR* on page AppxG-51
ITCMRR, ARMv6		*c9, TCM Region Registers (DTCMRR and ITCMRR)* on page AppxG-47
ITLBIALL	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
ITLBIASID	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
ITLBIMVA	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
ITLBLR, pre-ARMv7		*c10, VMSA TLB lockdown support* on page AppxH-59
ITSTATE		*ITSTATE* on page A2-17
Jazelle ID		*Jazelle ID Register (JIDR)* on page A2-76
Jazelle Main Configuration		*Jazelle Main Configuration Register (JMCR)* on page A2-77
Jazelle OS Control		*Jazelle OS Control Register (JOSCR)* on page B1-77
JIDR		*Jazelle ID Register (JIDR)* on page A2-76
JMCR		*Jazelle Main Configuration Register (JMCR)* on page A2-77
JOSCR		*Jazelle OS Control Register (JOSCR)* on page B1-77
Lock Access		*Lock Access Register (DBGLAR)* on page C10-94
Lock Status		*Lock Status Register (DBGLSR)* on page C10-95
LR		*ARM core registers* on page A2-11 for application-level description *ARM core registers* on page B1-9 for system-level description
LR_abt, LR_fiq, LR_irq, LR_mon, LR-_svc, LR-_und, LR_usr		*ARM core registers* on page B1-9

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
ITLBIASID	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
ITLBIMVA	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
ITLBLR、ARMv7 以前		*c10、VMSA TLB ロックダウンのサポート* P. 付録 H-59
ITSTATE		*ITSTATE* P. A2-17
Jazelle ID		*JazelleID レジスタ (JIDR)* P. A2-76
Jazelle メイン構成		*Jazelle メイン構成レジスタ (JMCR)* P. A2-77
Jazelle OS 制御		*Jazelle OS 制御レジスタ (JOSCR)* P. B1-77
JIDR		*JazelleID レジスタ (JIDR)* P. A2-76
JMCR		*Jazelle メイン構成レジスタ (JMCR)* P. A2-77
JOSCR		*Jazelle OS 制御レジスタ (JOSCR)* P. B1-77
ロックアクセス		*ロックアクセス レジスタ (DBGLAR)* P. C10-94
ロックステータス		*ロックステータス レジスタ (DBGLSR)* P. C10-95
LR		アプリケーションレベルの説明については、*ARM コアレジスタ* P. A2-11 を参照 システムレベルの説明については、*ARM コアレジスタ* P. B1-9 を参照
LR_abt、LR_fiq、LR_irq、LR_mon、LR-_svc、LR-_und、LR_usr		*ARM コアレジスタ* P. B1-9
メイン ID	PMSA	*c0、メイン ID レジスタ (MIDR)* P. B4-32
	VMSA	*c0、メイン ID レジスタ (MIDR)* P. B3-81
メディアおよび VFP 機能		*メディアおよび VFP 機能レジスタ* P. B5-36
メモリモデル機能		*CP15 c0、メモリモデル機能レジスタ* P. B5-9
メモリ領域アクセス許可、ARMv6 以前		*c5、メモリ領域アクセス許可レジスタ （DAPR および IAPR）* P. 付録 H-45
メモリ領域バッファ可能、ARMv6 以前		*c3、メモリ領域バッファ可能レジスタ (DBR)* P. 付録 H-44
メモリ領域キャッシュ可能、ARMv6 以前		*c2、メモリ領域キャッシュ可能レジスタ （DCR および ICR）* P. 付録 H-44

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
Main ID	PMSA	*c0, Main ID Register (MIDR)* on page B4-32
	VMSA	*c0, Main ID Register (MIDR)* on page B3-81
Media and VFP Feature		*Media and VFP Feature registers* on page B5-36
Memory Model Feature		*CP15 c0, Memory Model Feature registers* on page B5-9
Memory Region Access Permissions, pre-ARMv6		*c5, Memory Region Access Permissions Registers (DAPR and IAPR)* on page AppxH-45
Memory Region Bufferability, pre-ARMv6		*c3, Memory Region Bufferability Register (DBR)* on page AppxH-44
Memory Region Cacheability, pre-ARMv6		*c2, Memory Region Cacheability Registers (DCR and ICR)* on page AppxH-44
Memory Region, pre-ARMv6		*c6, Memory Region registers (DMRR0-DMRR7 and IMRR0-IMRR7)* on page AppxH-47
Memory Remap	VMSA	*CP15 c10, Memory Remap Registers* on page B3-143
MIDR	PMSA	*c0, Main ID Register (MIDR)* on page B4-32
	VMSA	*c0, Main ID Register (MIDR)* on page B3-81
Monitor Vector Base Address	VMSA	*c12, Monitor Vector Base Address Register (MVBAR)* on page B3-149
MPIDR	PMSA	*c0, Multiprocessor Affinity Register (MPIDR)* on page B4-37
	VMSA	*c0, Multiprocessor Affinity Register (MPIDR)* on page B3-87
MPU Region Number	PMSA	*c6, MPU Region Number Register (RGNR)* on page B4-66
MPU Type	PMSA	*c0, MPU Type Register (MPUIR)* on page B4-36
MPUIR	PMSA	*c0, MPU Type Register (MPUIR)* on page B4-36
Multiprocessor affinity	PMSA	*c0, Multiprocessor Affinity Register (MPIDR)* on page B4-37
	VMSA	*c0, Multiprocessor Affinity Register (MPIDR)* on page B3-87
MVBAR	VMSA	*c12, Monitor Vector Base Address Register (MVBAR)* on page B3-149
MVFR0		*Media and VFP Feature Register 0 (MVFR0)* on page B5-36
MVFR1		*Media and VFP Feature Register 1 (MVFR1)* on page B5-38

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
メモリ領域、ARMv6 以前		*c6、メモリ領域レジスタ （DMRR0 ～ DMRR7 および IMRR0 ～ IMRR7）* P. 付録 H-47
メモリ再マップ	VMSA	*CP15 c10、メモリ再マップレジスタ* P. B3-143
MIDR	PMSA	*c0、メイン ID レジスタ (MIDR)* P. B4-32
	VMSA	*c0、メイン ID レジスタ (MIDR)* P. B3-81
モニタベクタ ベース アドレス	VMSA	*c12、モニタベクタ ベースアドレス レジスタ (MVBAR)* P. B3-149
MPIDR	PMSA	*c0、マルチプロセッサ類似性レジスタ (MPIDR)* P. B4-37
	VMSA	*c0、マルチプロセッサ類似性レジスタ (MPIDR)* P. B3-87
MPU 領域番号	PMSA	*c6、MPU 領域番号レジスタ (RGNR)* P. B4-66
MPU タイプ	PMSA	*c0、MPU タイプレジスタ (MPUIR)* P. B4-36
MPUIR	PMSA	*c0、MPU タイプレジスタ (MPUIR)* P. B4-36
マルチプロセッサ類似性	PMSA	*c0、マルチプロセッサ類似性レジスタ (MPIDR)* P. B4-37
	VMSA	*c0、マルチプロセッサ類似性レジスタ (MPIDR)* P. B3-87
MVBAR	VMSA	*c12、モニタベクタ ベースアドレス レジスタ (MVBAR)* P. B3-149
MVFR0		*メディアおよび VFP 機能レジスタ 0 (MVFR0)* P. B5-36
MVFR1		*メディアおよび VFP 機能レジスタ 1(MVFR1)* P. B5-38
NMRR	VMSA	*c10、ノーマルメモリ再マップレジスタ (NMRR)* P. B3-146
非セキュアアクセス制御	VMSA	*c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3-110
非セキュアアクセス制御、ARMv6 での相違点		*c1、VMSA セキュリティ拡張機能のサポート* P. 付録 G-35
ノーマルメモリ再マップ	VMSA	*c10、ノーマルメモリ再マップレジスタ (NMRR)* P. B3-146
NSACR	VMSA	*c1、非セキュアアクセス制御レジスタ (NSACR)* P. B3-110
	ARMv6	*c9、TCM 非セキュアアクセス制御レジスタ、DTCM-NSACR および ITCM-NSACR* P. 付録 G-51
OS ロックアクセス		*OS ロックアクセス レジスタ (DBGOSLAR)* P. C10-75

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
NMRR	VMSA	*c10, Normal Memory Remap Register (NMRR)* on page B3-146
Non-secure Access Control	VMSA	*c1, Non-Secure Access Control Register (NSACR)* on page B3-110
Non-Secure Access Control, ARMv6 differences		*c1, VMSA Security Extensions support* on page AppxG-35
Normal Memory Remap	VMSA	*c10, Normal Memory Remap Register (NMRR)* on page B3-146
NSACR	VMSA	*c1, Non-Secure Access Control Register (NSACR)* on page B3-110
	ARMv6	*c9, TCM Non-Secure Access Control Registers, DTCM-NSACR and ITCM-NSACR* on page AppxG-51
OS Lock Access		*OS Lock Access Register (DBGOSLAR)* on page C10-75
OS Lock Status		*OS Lock Status Register (DBGOSLSR)* on page C10-76
OS Save and Restore		*OS Save and Restore Register (DBGOSSRR)* on page C10-77
Overflow Flag Status		*c9, Overflow Flag Status Register (PMOVSR)* on page C10-110
PAR	VMSA	*c7, Physical Address Register (PAR) and VA to PA translations* on page B3-133
PC		*ARM core registers* on page A2-11 for application-level description *ARM core registers* on page B1-9 for system-level description
Performance Monitor Control		*c9, Performance Monitor Control Register (PMCR)* on page C10-105
Peripheral ID		*Debug Peripheral Identification Registers (DBGPID0 to DBGPID4)* on page C10-98
PFF	PMSA	*Instruction Synchronization Barrier operation* on page B4-73
	VMSA	*Instruction Synchronization Barrier operation* on page B3-137
Physical Address	VMSA	*c7, Physical Address Register (PAR) and VA to PA translations* on page B3-133
PMCCNTR		*c9, Cycle Count Register (PMCCNTR)* on page C10-114
PMCNTENCLR		*c9, Count Enable Clear Register (PMCNTENCLR)* on page C10-109
PMCNTENSET		*c9, Count Enable Set Register (PMCNTENSET)* on page C10-108
PMCR		*c9, Performance Monitor Control Register (PMCR)* on page C10-105

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
OS ロックステータス		*OS ロックステータス レジスタ (DBGOSLSR)* P. C10-76
OS 保存および復元		*OS 保存および復元レジスタ (DBGOSSRR)* P. C10-77
オーバフローフラグ ステータス		*c9、オーバフローフラグステータス レジスタ (PMOVSR)* P. C10-110
PAR	VMSA	*c7、物理アドレスレジスタ (PAR) と VA から PA への変換* P. B3-133
PC		アプリケーションレベルの説明については、*ARM コアレジスタ* P. A2-11 を参照 システムレベルの説明については、*ARM コアレジスタ* P. B1-9 を参照
パフォーマンスモニタ制御		*c9、パフォーマンスモニタ制御レジスタ (PMCR)* P. C10-105
ペリフェラル ID		*デバッグペリフェラル識別レジスタ (DBGPID0 ～ DBGPID4)* P. C10-98
PFF	PMSA	*命令同期バリア操作* P. B4-73
	VMSA	*命令同期バリア操作* P. B3-137
物理アドレス	VMSA	*c7、物理アドレスレジスタ (PAR) と VA から PA への変換* P. B3-133
PMCCNTR		*c9、サイクルカウント レジスタ (PMCCNTR)* P. C10-114
PMCNTENCLR		*c9、カウントイネーブルクリア レジスタ (PMCNTENCLR)* P. C10-109
PMCNTENSET		*c9、カウントイネーブルセット レジスタ (PMCNTENSET)* P. C10-108
PMCR		*c9、パフォーマンスモニタ制御レジスタ (PMCR)* P. C10-105
PMINTENCLR		*c9、割り込みイネーブルクリア レジスタ (PMINTENCLR)* P. C10-119
PMINTENSET		*c9、割り込みイネーブルセット レジスタ (PMINTENSET)* P. C10-118
PMOVSR		*c9、オーバフローフラグステータス レジスタ (PMOVSR)* P. C10-110
PMSELR		*c9、イベントカウンタ選択レジスタ (PMSELR)* P. C10-113

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
PMINTENCLR		*c9, Interrupt Enable Clear Register (PMINTENCLR)* on page C10-119
PMINTENSET		*c9, Interrupt Enable Set Register (PMINTENSET)* on page C10-118
PMOVSR		*c9, Overflow Flag Status Register (PMOVSR)* on page C10-110
PMSELR		*c9, Event Counter Selection Register (PMSELR)* on page C10-113
PMSWINC		*c9, Software Increment Register (PMSWINC)* on page C10-112
PMUSERENR		*c9, User Enable Register (PMUSERENR)* on page C10-117
PMXEVCNTR		*c9, Event Count Register (PMXEVCNTR)* on page C10-116
PMXEVTYPER		*c9, Event Type Select Register (PMXEVTYPER)* on page C10-115
Power-down and Reset Control		*Device Power-down and Reset Control Register (DBGPRCR), v7 Debug only* on page C10-31
Power-down and Reset Status		*Device Power-down and Reset Status Register (DBGPRSR), v7 Debug only* on page C10-34
Prefetch Status, ARMv6		*c7, Block Transfer Status Register* on page AppxG-43
Primary Region Remap	VMSA	*c10, Primary Region Remap Register (PRRR)* on page B3-143
Processor Feature		*CP15 c0, Processor Feature registers* on page B5-4
Program Counter Sampling, Debug		*Program Counter Sampling Register (DBGPCSR)* on page C10-38
PRRR	VMSA	*c10, Primary Region Remap Register (PRRR)* on page B3-143
PSR		*Program Status Registers (PSRs)* on page B1-14
Q0 - Q15		*Advanced SIMD and VFP extension registers* on page A2-21
R0 - R15		*ARM core registers* on page A2-11 for application-level description *ARM core registers* on page B1-9 for system-level description
R0_usr - R12_usr		*ARM core registers* on page B1-9
R8_fiq - R12_fiq		*ARM core registers* on page B1-9
RGNR	PMSA	*c6, MPU Region Number Register (RGNR)* on page B4-66
Run Control, Debug		*Debug Run Control Register (DBGDRCR), v7 Debug only* on page C10-29
S0 - S31		*Advanced SIMD and VFP extension registers* on page A2-21

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
PMSWINC		*c9、ソフトウェアインクリメント レジスタ (PMSWINC)* P. C10-112
PMUSERENR		*c9、ユーザイネーブル レジスタ (PMUSERENR)* P. C10-117
PMXEVCNTR		*c9、イベントカウント レジスタ (PMXEVCNTR)* P. C10-116
PMXEVTYPER		*c9、イベントタイプ選択レジスタ (PMXEVTYPER)* P. C10-115
電力オフおよびリセット制御		*デバイス電力オフおよびリセット制御レジスタ (DBGPRCR)、v7 デバッグのみ* P. C10-31
電力オフおよびリセットステータス		*デバイス電力オフおよびリセットステータス レジスタ (DBGPRSR)、v7 デバッグのみ* P. C10-34
プリフェッチステータス、ARMv6		*c7、ブロック転送ステータスレジスタ* P. 付録 G-43
1 次領域再マップ	VMSA	*c10、1 次領域再マップレジスタ (PRRR)* P. B3-143
プロセッサ機能		*CP15 c0、プロセッサ機能レジスタ* P. B5-4
プログラムカウンタ サンプリング、デバッグ		*プログラムカウンタサンプリング レジスタ (DBGPCSR)* P. C10-38
PRRR	VMSA	*c10、1 次領域再マップレジスタ (PRRR)* P. B3-143
PSR		*プログラムステータス レジスタ (PSR)* P. B1-14
Q0 ～ Q15		*アドバンスト SIMD と VFP の拡張機能レジスタ* P. A2-21
R0 ～ R15		アプリケーションレベルの説明については、*ARM コアレジスタ* P. A2-11 を参照 システムレベルの説明については、*ARM コアレジスタ* P. B1-9 を参照
R0_usr ～ R12_usr		*ARM コアレジスタ* P. B1-9
R8_fiq ～ R12_fiq		*ARM コアレジスタ* P. B1-9
RGNR	PMSA	*c6、MPU 領域番号レジスタ (RGNR)* P. B4-66
実行制御、デバッグ		*デバッグ実行制御レジスタ (DBGDRCR)、v7 デバッグのみ* P. C10-29
S0 ～ S31		*アドバンスト SIMD と VFP の拡張機能レジスタ* P. A2-21
SCR	VMSA	*c1、セキュア構成レジスタ (SCR)* P. B3-106

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
SCR	VMSA	*c1, Secure Configuration Register (SCR)* on page B3-106
SCTLR	PMSA	*c1, System Control Register (SCTLR)* on page B4-45
	VMSA	*c1, System Control Register (SCTLR)* on page B3-96
SDER	VMSA	*c1, Secure Debug Enable Register (SDER)* on page B3-108
Secure Configuration	VMSA	*c1, Secure Configuration Register (SCR)* on page B3-106
Secure Debug Enable	VMSA	*c1, Secure Debug Enable Register (SDER)* on page B3-108
Software Increment		*c9, Software Increment Register (PMSWINC)* on page C10-112
Software Thread ID	PMSA	*CP15 c13 Software Thread ID registers* on page B4-77
	VMSA	*CP15 c13 Software Thread ID registers* on page B3-154
SP		*ARM core registers* on page A2-11 for application-level description *ARM core registers* on page B1-9 for system-level description
SP-_abt, SP_fiq, SP_irq, SP_mon, SP-_svc, SP-_und, SP_usr		*ARM core registers* on page B1-9
SPSR		*The Saved Program Status Registers (SPSRs)* on page B1-15
SPSR_abt, SPSR_fiq, SPSR_irq, SPSR_mon, SPSR-_svc, SPSR-_und		*ARM core registers* on page B1-9
System Control	PMSA	*CP15 c1, System control registers* on page B4-44
System Control	VMSA	*CP15 c1, System control registers* on page B3-96
Target to Host Data Transfer		*Target to Host Data Transfer Register (DBGDTRTX)* on page C10-43
TCM Data Region, ARMv6		*c9, TCM Region Registers (DTCMRR and ITCMRR)* on page AppxG-47
TCM Instruction or unified Region, ARMv6		*c9, TCM Region Registers (DTCMRR and ITCMRR)* on page AppxG-47
TCM Non-Secure Access Control, ARMv6		*c9, TCM Non-Secure Access Control Registers, DTCM-NSACR and ITCM-NSACR* on page AppxG-51
TCM Selection, ARMv6		*c9, TCM Selection Register (TCMSR)* on page AppxG-46
TCM Type	PMSA	*c0, TCM Type Register (TCMTR)* on page B4-35
	VMSA	*c0, TCM Type Register (TCMTR)* on page B3-85

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
SCTLR	PMSA	*c1、システム制御レジスタ (SCTLR)* P. B4-45
	VMSA	*c1、システム制御レジスタ (SCTLR)* P. B3-96
SDER	VMSA	*c1、セキュアデバッグ イネーブルレジスタ (SDER)* P. B3-108
セキュア構成	VMSA	*c1、セキュア構成レジスタ (SCR)* P. B3-106
セキュアデバッグ イネーブル	VMSA	*c1、セキュアデバッグ イネーブルレジスタ (SDER)* P. B3-108
ソフトウェアインクリメント		*c9、ソフトウェアインクリメント レジスタ (PMSWINC)* P. C10-112
ソフトウェアスレッド ID	PMSA	*CP15 c13、ソフトウェアスレッド ID レジスタ* P. B4-77
	VMSA	*CP15 c13、ソフトウェアスレッド ID レジスタ* P. B3-154
SP		アプリケーションレベルの説明については、*ARM コアレジスタ* P. A2-11 を参照 システムレベルの説明については、*ARM コアレジスタ* P. B1-9 を参照
SP-_abt、SP_fiq、SP_irq、SP_mon、SP-_svc、SP-_und、SP_usr		*ARM コアレジスタ* P. B1-9
SPSR		*保存プログラム ステータスレジスタ (SPSR)* P. B1-15
SPSR_abt、SPSR_fiq、SPSR_irq、SPSR_mon、SPSR-_svc、SPSR-_und		*ARM コアレジスタ* P. B1-9
システム制御	PMSA	*CP15 c1、システム制御レジスタ* P. B4-44
システム制御	VMSA	*CP15 c1、システム制御レジスタ* P. B3-96
ターゲットからホストへのデータ転送		*ターゲットからホストへのデータ転送レジスタ (DBGDTRTX)* P. C10-43
TCM データ領域、ARMv6		*c9、TCM 領域レジスタ (DTCMRR および ITCMRR)* P. 付録 G-47
TCM 命令または統一領域、ARMv6		*c9、TCM 領域レジスタ (DTCMRR および ITCMRR)* P. 付録 G-47
TCM 非セキュアアクセス制御、ARMv6		*c9、TCM 非セキュアアクセス制御レジスタ、DTCM-NSACR および ITCM-NSACR* P. 付録 G-51
TCM 選択、ARMv6		*c9、TCM 選択レジスタ (TCMSR)* P. 付録 G-46

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
TCMSR, ARMv6		*c9, TCM Selection Register (TCMSR)* on page AppxG-46
TCMTR	PMSA	*c0, TCM Type Register (TCMTR)* on page B4-35
	VMSA	*c0, TCM Type Register (TCMTR)* on page B3-85
TEECR		*ThumbEE Configuration Register (TEECR)* on page A2-70
TEEHBR		*ThumbEE Handler Base Register (TEEHBR)* on page A2-71
TEX Remap	VMSA	*CP15 c10, Memory Remap Registers* on page B3-143
ThumbEE Configuration		*ThumbEE Configuration Register (TEECR)* on page A2-70
ThumbEE Handler Base		*ThumbEE Handler Base Register (TEEHBR)* on page A2-71
TLB Lockdown Register, pre-ARMv7		*c10, VMSA TLB lockdown support* on page AppxH-59
TLB Type	VMSA	*c0, TLB Type Register (TLBTR)* on page B3-86
TLBIALL	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
TLBIALLIS	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
TLBIASID	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
TLBIASIDIS	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
TLBIMVA	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
TLBIMVAA	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
TLBIMVAAIS	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
TLBIMVAIS	VMSA	*CP15 c8, TLB maintenance operations* on page B3-138
TLBTR		*c0, TLB Type Register (TLBTR)* on page B3-86
TPIDRPRW	PMSA	*CP15 c13 Software Thread ID registers* on page B4-77
	VMSA	*CP15 c13 Software Thread ID registers* on page B3-154
TPIDRURO	PMSA	*CP15 c13 Software Thread ID registers* on page B4-77
	VMSA	*CP15 c13 Software Thread ID registers* on page B3-154
TPIDRURW	PMSA	*CP15 c13 Software Thread ID registers* on page B4-77
	VMSA	*CP15 c13 Software Thread ID registers* on page B3-154

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
TCM タイプ	PMSA	*c0、TCM タイプレジスタ (TCMTR)* P. B4-35
	VMSA	*c0、TCM タイプレジスタ (TCMTR)* P. B3-85
TCMSR、ARMv6		*c9、TCM 選択レジスタ (TCMSR)* P. 付録 G-46
TCMTR	PMSA	*c0、TCM タイプレジスタ (TCMTR)* P. B4-35
	VMSA	*c0、TCM タイプレジスタ (TCMTR)* P. B3-85
TEECR		*ThumbEE 構成レジスタ (TEECR)* P. A2-70
TEEHBR		*ThumbEE ハンドラ ベースレジスタ (TEEHBR)* P. A2-71
TEX 再マップ	VMSA	*CP15 c10、メモリ再マップレジスタ* P. B3-143
ThumbEE 構成		*ThumbEE 構成レジスタ (TEECR)* P. A2-70
ThumbEE ハンドラベース		*ThumbEE ハンドラ ベースレジスタ (TEEHBR)* P. A2-71
TLB ロックダウンレジスタ、ARMv7 以前		*c10、VMSA TLB ロックダウンのサポート* P. 付録 H-59
TLB タイプ	VMSA	*c0、TLB タイプレジスタ (TLBTR)* P. B3-86
TLBIALL	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
TLBIALLIS	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
TLBIASID	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
TLBIASIDIS	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
TLBIMVA	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
TLBIMVAA	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
TLBIMVAAIS	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
TLBIMVAIS	VMSA	*CP15 c8、TLB 保守操作* P. B3-138
TLBTR		*c0、TLB タイプレジスタ (TLBTR)* P. B3-86
TPIDRPRW	PMSA	*CP15 c13、ソフトウェアスレッド ID レジスタ* P. B4-77
	VMSA	*CP15 c13、ソフトウェアスレッド ID レジスタ* P. B3-154

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
Translation Table Base	VMSA	*CP15 c2, Translation table support registers* on page B3-113
TTBCR	VMSA	*c2, Translation Table Base Control Register (TTBCR)* on page B3-117
TTBR0	VMSA	*c2, Translation Table Base Register 0 (TTBR0)* on page B3-113
TTBR1	VMSA	*c2, Translation Table Base Register 1 (TTBR1)* on page B3-116
User Enable		*c9, User Enable Register (PMUSERENR)* on page C10-117
UTLBIALL	VMSA	Previous names for the CP15 c8 operations TLBIALL, TLBIASID, and TLBIMVA, see *CP15 c8, TLB maintenance operations* on page B3-138
UTLBIASID	VMSA	
UTLBIMVA	VMSA	
V2PCWPR	VMSA	*CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130
V2PCWPW	VMSA	*CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130
V2PCWUR	VMSA	*CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130
V2PCWUW	VMSA	*CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130
V2POWPR	VMSA	*CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130
V2POWPW	VMSA	*CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130
V2POWUR	VMSA	*CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130
V2POWUW	VMSA	*CP15 c7, Virtual Address to Physical Address translation operations* on page B3-130
VBAR	VMSA	*c12, Vector Base Address Register (VBAR)* on page B3-148
Vector Base Address	VMSA	*c12, Vector Base Address Register (VBAR)* on page B3-148
Vector Catch Register		*Vector Catch Register (DBGVCR)* on page C10-67
Watchpoint Control		*Watchpoint Control Registers (DBGWCR)* on page C10-61

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
TPIDRURO	PMSA	*CP15 c13、ソフトウェアスレッド ID レジスタ* P. B4-77
	VMSA	*CP15 c13、ソフトウェアスレッド ID レジスタ* P. B3-154
TPIDRURW	PMSA	*CP15 c13、ソフトウェアスレッド ID レジスタ* P. B4-77
	VMSA	*CP15 c13、ソフトウェアスレッド ID レジスタ* P. B3-154
変換テーブルベース	VMSA	*CP15 c2、変換テーブルサポート レジスタ* P. B3-113
TTBCR	VMSA	*c2、変換テーブルベース制御レジスタ (TTBCR)* P. B3-117
TTBR0	VMSA	*c2、変換テーブルベースレジスタ 0 (TTBR0)* P. B3-113
TTBR1	VMSA	*c2、変換テーブルベースレジスタ 1(TTBR1)* P. B3-116
ユーザイネーブル		*c9、ユーザイネーブル レジスタ (PMUSERENR)* P. C10-117
UTLBIALL	VMSA	CP15 c8 の操作 TLBIALL、TLBIASID、TLBIMVA の以前の名称については、*CP15 c8、TLB 保守操作* P. B3-138 を参照
UTLBIASID	VMSA	
UTLBIMVA	VMSA	
V2PCWPR	VMSA	*CP15 c7、仮想アドレスから物理アドレスへの変換操作* P. B3-130
V2PCWPW	VMSA	*CP15 c7、仮想アドレスから物理アドレスへの変換操作* P. B3-130
V2PCWUR	VMSA	*CP15 c7、仮想アドレスから物理アドレスへの変換操作* P. B3-130
V2PCWUW	VMSA	*CP15 c7、仮想アドレスから物理アドレスへの変換操作* P. B3-130
V2POWPR	VMSA	*CP15 c7、仮想アドレスから物理アドレスへの変換操作* P. B3-130
V2POWPW	VMSA	*CP15 c7、仮想アドレスから物理アドレスへの変換操作* P. B3-130
V2POWUR	VMSA	*CP15 c7、仮想アドレスから物理アドレスへの変換操作* P. B3-130
V2POWUW	VMSA	*CP15 c7、仮想アドレスから物理アドレスへの変換操作* P. B3-130
VBAR	VMSA	*c12、ベクタ ベースアドレスレジスタ (VBAR)* P. B3-148

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Table K-1 Register index (continued)

Register	In[a]	Description, see
Watchpoint Fault Address, CP14		*Watchpoint Fault Address Register (DBGWFAR)* on page C10-28
Watchpoint Fault Address, CP15, ARMv6		*c6, Watchpoint Fault Address Register (DBGWFAR)* on page AppxG-37
Watchpoint Value		*Watchpoint Value Registers (DBGWVR)* on page C10-60

a. Applies only to entries for ARMv7 CP15 registers and operations. Where these are included in both a VMSA implementation and a PMSA implementation these are described in Chapter B3 *Virtual Memory System Architecture (VMSA)* and in Chapter B4 *Protected Memory System Architecture (PMSA)*, and both descriptions are included in this index.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

表 K-1 レジスタ索引 （続き）

レジスタ	実装[a]	説明、参照先
ベクタ ベースアドレス	VMSA	*c12*、*ベクタ ベースアドレスレジスタ (VBAR)* P. B3-148
ベクタキャッチ レジスタ		*ベクタキャッチ レジスタ (DBGVCR)* P. C10-67
ウォッチポイント制御		*ウォッチポイント制御レジスタ (DBGWCR)* P. C10-61
ウォッチポイント フォールト アドレス、CP14		*ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. C10-28
ウォッチポイント フォールト アドレス、CP15、ARMv6		*c6*、*ウォッチポイントフォールトアドレス レジスタ (DBGWFAR)* P. 付録 G-37
ウォッチポイント値		*ウォッチポイント値レジスタ (DBGWVR)* P. C10-60

a. ARMv7 の CP15 レジスタおよび操作のエントリにのみ適用されます。VMSA 実装と PMSA 実装の両方に含まれる場合については第 B3 章*「仮想メモリシステム アーキテクチャ (VMSA)」*と第 B4 章*「保護メモリシステム アーキテクチャ (PMSA)」*に説明があり、どちらの説明もこの索引に記載されています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Glossary

Abort Is caused by an illegal memory access. Aborts can be caused by the external memory system or the MMU or MPU.

Abort model
Describes what happens to the processor state when a Data Abort exception occurs. Different abort models behave differently with regard to load/store instructions that specify base register write-back. For more details, see *Effects of data-aborted instructions* on page B1-57.

Addressing mode
Means a method for generating the memory address used by a load/store instruction.

Advanced SIMD
Is an extension to the ARM architecture that provides SIMD operations on a bank of extension registers. If the VFP extension is also implemented, the two extensions share the register bank and the SIMD operations include single-precision floating-point SIMD operations.

Aligned Refers to data items stored in such a way that their address is divisible by the highest power of 2 that divides their size. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of an element of the access

ARM instruction
Is a word that specifies an operation for a processor in ARM state to perform. ARM instructions must be word-aligned.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

用語集

0 への丸め (RZ) モード
丸め結果が、丸め前の結果よりも絶対値が大きくない、最も近い表現可能な数値となることを意味します。

ARM 命令
ARM 状態のプロセッサが実行する動作を指定するワード。ARM 命令はワードアラインしている必要があります。

DNM 参照先：変更不可

DSP 参照先：デジタル信号処理。

FCSE 参照先：高速コンテキストスイッチ拡張機能。

Flush-to-Zero モード
非正規化オペランドと中間結果を 0 で置き換え、最終結果の正確性に大きな影響を及ぼすことなく一部の VFP アルゴリズムのパフォーマンスを最適化する、特別な処理モード。

IMP 該当するビットの動作が実装定義であることを示すため、図中で使用される略号。

MMU 参照先：メモリ管理ユニット。

MPU 参照先：メモリ保護ユニット。

MVA 参照先：修飾仮想アドレス。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Atomicity
Is a term that describes either single-copy atomicity or multi-copy atomicity. The forms of atomicity used in the ARM architecture are defined in *Atomicity in the ARM architecture* on page A3-26.

See also Multi-copy Atomicity, Single-copy atomicity.

Banked register
Is a register that has multiple instances, with the instance that is in use depending on the processor mode, security state, or other processor state.

Base register
Is a register specified by a load/store instruction that is used as the base value for the instruction's address calculation. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.

Base register write-back
Describes writing back a modified value to the base register used in an address calculation.

Big-endian memory
Means that:

- a byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address
- a byte at a halfword-aligned address is the most significant byte in the halfword at that address.

Blocking
Describes an operation that does not permit following instructions to be executed before the operation is completed.

A non-blocking operation can permit following instructions to be executed before the operation is completed, and in the event of encountering an exception do not signal an exception to the processor. This enables implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise processor state.

Branch prediction
Is where a processor chooses a future execution path to prefetch along (see Prefetching). For example, after a branch instruction, the processor can choose to prefetch either the instruction following the branch or the instruction at the branch target.

Breakpoint
Is a debug event triggered by the execution of a particular instruction, specified in terms of the address of the instruction and/or the state of the processor when the instruction is executed.

Byte Is an 8-bit data item.

Cache Is a block of high-speed memory locations whose addresses are changed automatically in response to which memory locations the processor is accessing, and whose purpose is to increase the average speed of a memory access.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

NaN NaN は浮動小数点数の特別な値で、数値と無限大のいずれも適切でない場合に使用されます。NaN には、ほとんどの浮動小数点演算で伝播される *クワイエット* NaN と、使用された場合に無効演算浮動小数点例外を引き起こす *シグナル* NaN があります。詳細については、IEEE 754 規格を参照して下さい。

PA 参照先：物理アドレス。

RAO 参照先：読み出し値 1。

RAO/SBOP
読み出し値 1、書き込み時には 1 または保持。

このビットはどの実装でも 1（ビットフィールドの場合はすべてのビットが 1）として読み出され、フィールドへの書き込みは無視される必要があります。

ソフトウェアは、このフィールドからの読み出し値が 1（またはすべて 1）であることを前提とできますが、書き込み時には SBOP ポリシーを守る必要があります。

RAO/WI 読み出し値 1、書き込み無視。

このビットはどの実装でも 1（ビットフィールドの場合はすべてのビットが 1）として読み出され、フィールドへの書き込みは無視される必要があります。

ソフトウェアは、このフィールドからの読み出し値が 1（またはすべて 1）であり、書き込みは無視されることを前提とできます。

RAZ 参照先：読み出し値 0。

RAZ/SBZP
読み出し値 0、書き込み時には 0 または保持。

このビットはどの実装でも 0（ビットフィールドの場合はすべてのビットが 0）として読み出され、フィールドへの書き込みは無視される必要があります。

ソフトウェアは、このフィールドからの読み出し値が 0（またはすべて 0）であることを前提とできますが、書き込み時には SBZP ポリシーを守る必要があります。

RAZ/WI 読み出し値 0、書き込み無視。

このビットはどの実装でも 0（ビットフィールドの場合はすべてのビットが 0）として読み出され、フィールドへの書き込みは無視される必要があります。

ソフトウェアは、このフィールドからの読み出し値が 0（またはすべて 0）であり、書き込みは無視されることを前提とできます。

RISC 縮小命令セットコンピュータの略。

SBO 参照先：常に 1(Should Be One)。

SBOP 参照先：常に 1 または保持。

SBZ 参照先：常に 0(Should Be Zero)。

SBZP 参照先：常に 0 または保持。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Cache contention
Is when the number of frequently-used memory cache lines that use a particular cache set exceeds the set-associativity of the cache. In this case, main memory activity goes up and performance drops.

Cache hit
Is a memory access that can be processed at high speed because the data it addresses is already in the cache.

Cache line
Is the basic unit of storage in a cache. Its size is always a power of two (usually 4 or 8 words), and must be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely just called cache lines.

Cache line index
Is a number associated with each cache line in a cache set. In each cache set, the cache lines are numbered from 0 to (set associativity)–1.

Cache lockdown
Alleviates the delays caused by accessing a cache in a worst-case situation. Cache lockdown enables critical code and data to be loaded into the cache so that the cache lines containing them are not subsequently re-allocated. This ensures that all subsequent accesses to the code and data concerned are cache hits and so complete quickly.

Cache lockdown blocks
Consist of one line from each cache set. Cache lockdown is performed in units of a cache lockdown block.

Cache miss
Is a memory access that cannot be processed at high speed because the data it addresses is not in the cache.

Cache sets
Are areas of a cache, divided up to simplify and speed up the process of determining whether a cache hit occurs. The number of cache sets is always a power of two.

Cache way
A cache way consists of one cache line from each cache set. The cache ways are indexed from 0 to ASSOCIATIVITY-1. The cache lines in a cache way are chosen to have the same index as the cache way. So for example cache way 0 consists of the cache line with index 0 from each cache set, and cache way *n* consists of the cache line with index *n* from each cache set.

Callee-save registers
Are registers that a called procedure must preserve. To preserve a callee-save register, the called procedure would normally either not use the register at all, or store the register to the stack during procedure entry and re-load it from the stack during procedure exit.

Caller-save registers
Are registers that a called procedure need not preserve. If the calling procedure requires their values to be preserved, it must store and reload them itself.

Condition field
Is a 4-bit field in an instruction that is used to specify a condition under which the instruction can execute.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

SIMD　単一命令、複数データ操作の意味。

Thumb 命令
Thumb 状態のプロセッサが実行する動作を指定する、1 つまたは 2 つのハーフワード。Thumb 命令は、ハーフワードアラインドの必要があります。

TLB　参照先：変換ルックアサイドバッファ。

TLB ロックダウン
特定の変換テーブルウォークの結果がアクセスされることを防止する方法。これにより、関連するメモリ領域にアクセスする際に変換テーブルウォークが一切発生しなくなります。

UNK　ソフトウェアが、このフィールドに含まれている値を未知として扱わなければならないことを示す略語。

このビットは、すべての実装で 0（ビットフィールドの場合はすべてのビットが 0）として読み出される必要があります。ソフトウェアは、このフィールドの読み出し値が 0 であることを前提にできません。

UNK/SBOP
読み出し値未知、書き込み時には 1 または保持。

このビットはどの実装でも 1（ビットフィールドの場合はすべてのビットが 1）として読み出され、フィールドへの書き込みは無視される必要があります。

ソフトウェアは、このフィールドからの読み出し値が 1（ビットフィールドの場合はすべて 1）であることを前提にできません。書き込み時には SBOP ポリシーを守る必要があります。

UNK/SBZP
読み出し値未知、書き込み時には 0 または保持。

このビットはどの実装でも 0（ビットフィールドの場合はすべてのビットが 0）として読み出され、フィールドへの書き込みは無視される必要があります。

ソフトウェアは、このフィールドからの読み出し値が 0（ビットフィールドの場合はすべてのビットが 0）であることを前提にできません。書き込み時には SBZP ポリシーを守る必要があります。

VA　参照先：仮想アドレス。

VFP　ARM アーキテクチャのコプロセッサ拡張機能。単精度と倍精度の浮動小数点演算を提供します。

アドバンスト SIMD
ARM アーキテクチャの拡張機能で、拡張レジスタのバンク上で SIMD 演算を行います。VFP 拡張機能も実装されている場合、これら 2 つの拡張機能はレジスタバンクを共有し、SIMD 演算には単精度浮動小数点 SIMD 演算が含まれます。

アトミック性
シングルコピーのアトミック性、またはマルチコピーのアトミック性を意味する用語。ARM アーキテクチャで使用されるアトミック性の形式は、*ARM アーキテクチャにおけるアトミック性* P. A3-26 に定義されています。

参照先：マルチコピーのアトミック性、シングルコピーのアトミック性。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Conditional execution
Means that if the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing.

Configuration
Settings made on reset, or immediately after reset, and normally expected to remain static throughout program execution.

Context switch
Is the saving and restoring of computational state when switching between different threads or processes. In this manual, the term context switch is used to describe any situations where the context is switched by an operating system and might or might not include changes to the address space.

Data cache
Is a separate cache used only for processing data loads and stores.

Digital signal processing (DSP)
Refers to a variety of algorithms that are used to process signals that have been sampled and converted to digital form. Saturated arithmetic is often used in such algorithms.

Direct-mapped cache
Is a one-way set-associative cache. Each cache set consists of a single cache line, so cache look-up just needs to select and check one cache line.

Direct Memory Access
Is an operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

DNM *See* Do-not-modify.

Domain Is a collection of sections, Large pages and Small pages of memory, that can have their access permissions switched rapidly by writing to the Domain Access Control Register, in CP15 c3.

Do-not-modify (DNM)
Means the value must not be altered by software. DNM fields read as UNKNOWN values, and must only be written with the same value read from the same field on the same processor.

Double-precision value
Consists of two 32-bit words that must appear consecutively in memory and must both be word-aligned, and that is interpreted as a basic double-precision floating-point number according to the IEEE 754-1985 standard.

Doubleword
Is a 64-bit data item. Doublewords are normally at least word-aligned in ARM systems.

Doubleword-aligned
Means that the address is divisible by 8.

DSP *See* Digital signal processing

Endianness
Is an aspect of the system memory mapping. See big-endian and little-endian.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

アドレシングモード
ロード / ストア命令で使用されるメモリアドレスを計算する方法。

アボート 不正なメモリアクセスにより発生します。アボートは、外部メモリシステムや MMU または MPU により発生することがあります。

アボートモデル
データアボート例外が発生したときに、プロセッサ状態にどのような変化が起きるかを記述しています。アボートモデルによって、ベースレジスタのライトバックを指定するロード / ストア命令に関連した動作が異なります。詳細については、*データアボートされた命令の影響* P. B1-57 を参照して下さい。

アラインド
データ項目が、そのデータサイズを格納できる最小の 2 のべき乗の値で割り切れるようなアドレスに格納されている場合、そのデータ項目はアラインしていると呼びます。したがって、アラインしているハーフワード、ワード、ダブルワードのアドレスは、それぞれ 2、4、8 で割り切れます。

アラインドアクセスは、それがアクセスする要素のサイズにアドレスがアラインしているようなアクセスを指します。

アンアラインド
アンアラインドアクセスは、アクセスのアドレスがアクセスする要素のサイズにアラインしていないアクセスを指します。

アンアラインド メモリアクセス
適切にハーフワード、ワード、またはダブルワードにアラインしていない、またはその可能性があるメモリアクセス。

イミディエートおよびオフセットフィールド
これらのフィールドは、特記されていない限り符号なしです。

イミディエート値
命令に直接エンコードされ、命令の実行時に数値データとして使用される値。多くの ARM および Thumb 命令では、小さな数値をイミディエート値として、その数値を使用する命令にエンコードできます。

インターワーキング
ARM コードと Thumb コードとの間の分岐を許可するワーキング方法。

インデクスなしのアドレシング
ベースレジスタの値を直接仮想アドレスとしてメモリへ送信し、オフセットの加減算を行わないアドレシング。ほとんどのタイプのロード / ストア命令では、オフセットアドレシングのイミディエートオフセットに 0 を指定することで、インデクスなしのアドレシングを行います。
LDC、LDC2、STC、STC2 命令には明示的なインデクスなしのアドレシングモードがあり、命令のオフセットフィールドを追加のコプロセッサオプションの指定に使用できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Exception
Handles an event. For example, an exception could handle an external interrupt or an Undefined Instruction.

Exception modes
Are privileged modes that are entered when specific exceptions occur.

Exception vector
Is one of a number of fixed addresses in low memory, or in high memory if high vectors are configured.

Execution stream
The stream of instructions that would have been executed by sequential execution of the program.

Explicit access
A read from memory, or a write to memory, generated by a load/store instruction executed in the processor. Reads and writes generated by L1 DMA accesses or hardware translation table accesses are not explicit accesses.

External abort
Is an abort that is generated by the external memory system.

Fault Is an abort that is generated by the MMU.

Fast Context Switch Extension (FCSE)
Modifies the behavior of an ARM memory system to enable multiple programs running on the ARM processor to use identical address ranges, while ensuring that the addresses they present to the rest of the memory system differ. From ARMv6, use of the FCSE is deprecated, and the FCSE is optional in ARMv7.

FCSE *See* Fast Context Switch Extension.

Flat address mapping
Is where the physical address for every access is equal to its virtual address.

Flush-to-zero mode
Is a special processing mode that optimizes the performance of some VFP algorithms by replacing the denormalized operands and intermediate results with zeros, without significantly affecting the accuracy of their final results.

Fully-associative cache
Has just one cache set, that consists of the entire cache. See also direct-mapped cache.

General-purpose register
Is one of the 32-bit general-purpose integer registers, R0 to R15. Note that R15 holds the Program Counter, and there are often limitations on its use that do not apply to R0 to R14.

Halfword
Is a 16-bit data item. Halfwords are normally halfword-aligned in ARM systems.

Halfword-aligned
Means that the address is divisible by 2.

High registers
Are ARM core registers 8 to 15, that can be accessed by some Thumb instructions.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

インデクスレジスタ
一部のロード / ストア命令で指定されるレジスタ。このレジスタの値は、メモリに送出する仮想アドレスを生成するために、オフセットとしてベースレジスタの値に加算または減算されます。一部のアドレシングモードでは、必要に応じて、インデクスレジスタの値を加算または減算する前にシフトできます。

インラインリテラル
コード自体と同じ領域に保持される定数アドレス、およびその他のデータ項目。これらはコンパイラにより自動的に生成され、アセンブラのコードにも含まれていることがあります。

ウォッチポイント
メモリへのアクセスによってトリガされるデバッグイベントで、アクセスされるメモリ位置のアドレスによって定義されます。

エンディアン形式
システムのメモリマッピングの要素。ビッグエンディアンおよびリトルエンディアン参照。

エンディアン混在
ビッグエンディアンのデータとリトルエンディアンのデータへのアクセスを自由に混ぜることができ、それによってパフォーマンスの低下やコードサイズへのペナルティがわずかである場合、そのプロセッサはエンディアン混在メモリアクセスをサポートしていると呼びます。

オブザーバ
システム内の、ペリフェラルデバイスなどのプロセッサまたは機構で、メモリへの読み出し / 書き込みを生成できるもの。

オフセットアドレシング
メモリアドレスが、ベースレジスタの値にオフセットを加算または減算して計算されることを意味します。

外部アボート
外部メモリシステムにより生成されるアボート。

書き込み割り当てキャッシュ
データのストア時のキャッシュミスにより、キャッシュ内のキャッシュラインが割り当てられる方式のキャッシュ。

仮想アドレス (VA)
ARM プロセッサにより生成されるアドレス。PMSA 実装では、仮想アドレスは物理アドレスと同一です。

完全アソシエイティブキャッシュ
1 つのキャッシュセットのみでキャッシュ全体が構成されるキャッシュ。直接マップキャッシュも参照。

擬似命令
ある命令エンコードにアセンブルされるが、それとは別の異なるアセンブラ構文に逆アセンブルされることが期待される UAL アセンブラ構文で、本書ではその異なる構文で説明されています。例えば、`MOV <Rd>,<Rm>, LSL #<n>` は擬似命令で、`LSL <Rd>,<Rm>,#<n>` として逆アセンブルされることを前提としています。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

High vectors
Are alternative locations for exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.

Immediate and offset fields
Are unsigned unless otherwise stated.

Immediate values
Are values that are encoded directly in the instruction and used as numeric data when the instruction is executed. Many ARM and Thumb instructions permit small numeric values to be encoded as immediate values in the instruction that operates on them.

IMP Is an abbreviation used in diagrams to indicate that the bit or bits concerned have IMPLEMENTATION DEFINED behavior.

IMPLEMENTATION DEFINED
Means that the behavior is not architecturally defined, but should be defined and documented by individual implementations.

Index register
Is a register specified in some load/store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the virtual address that is sent to memory. Some addressing modes optionally permit the index register value to be shifted before the addition or subtraction.

Inline literals
These are constant addresses and other data items held in the same area as the code itself. They are automatically generated by compilers, and can also appear in assembler code.

Instruction cache
Is a separate cache used only for processing instruction fetches.

Interworking
Is a method of working that permits branches between ARM and Thumb code.

Little-endian memory
Means that:

- a byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address
- a byte at a halfword-aligned address is the least significant byte in the halfword at that address.

Load/Store architecture
Is an architecture where data-processing operations only operate on register contents, not directly on memory contents.

Long branch
Is the use of a load instruction to branch to anywhere in the 4GB address space.

Memory barrier
See *Memory barriers* on page A3-47.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

キャッシュ
高速なメモリのブロックで、プロセッサがアクセスするメモリ位置に応じて自動的にアドレスが変更されます。キャッシュはメモリアクセスの平均速度を向上することを目的としています。

キャッシュウェイ
キャッシュウェイは、各キャッシュセットから 1 つのキャッシュラインで構成されます。キャッシュウェイには、0 から（アソシエティビティ）－1 までのインデクスが付けられています。
キャッシュウェイのキャッシュラインは、キャッシュウェイと同じインデクスを持つように選択されます。このため、例えばキャッシュウェイ 0 は、各キャッシュセットでインデクス 0 を持つキャッシュラインで構成され、キャッシュウェイ *n* は、各キャッシュセットでインデクス *n* を持つキャッシュラインで構成されます。

キャッシュセット
キャッシュの領域で、キャッシュヒットが発生するかどうかを判定するプロセスを簡素化し、高速化するために分割されています。キャッシュセットの数は、必ず 2 のべき乗です。

キャッシュの競合
ある特定のキャッシュセットを使用する、頻繁に使用されるメモリ キャッシュラインの数が、キャッシュのセットアソシエティビティを超えたとき。この場合は、メイン メモリの動作が増大して、性能が低下します。

キャッシュヒット
アクセスされるデータがすでにキャッシュにあるため、高速で処理できるメモリアクセス。

キャッシュミス
アクセスされるデータがキャッシュに存在しないため、高速で処理できないメモリアクセス。

キャッシュライン
キャッシュ内の記憶域の基本単位。このサイズは常に 2 のべき乗（通常は 4 または 8 ワード）で、適切なメモリ境界にアラインしている必要があります。*メモリ キャッシュラインは*、キャッシュラインと同じサイズ、同じアライメントのメモリブロックです。メモリ キャッシュラインを大まかに、キャッシュラインとだけ呼ぶこともあります。

キャッシュライン インデクス
キャッシュセットの各キャッシュラインに関連付けられた番号。各キャッシュセットのキャッシュラインには、0 から（セットアソシエティビティ）－1 までの番号が付けられます。

キャッシュロックダウン
最悪の状況でキャッシュへのアクセスにより発生する遅延を軽減する方法。キャッシュロックダウンを使用して、重要なコードやデータをキャッシュにロードし、それらを保持するキャッシュラインが後から再割り当てされないようにすることができます。これにより、対象のコードやデータに対する以降のアクセスが必ずキャッシュヒットになるため、高速に実行されることが保証されます。

キャッシュロックダウン ブロック
各キャッシュセットから 1 つのラインで構成されます。キャッシュロックダウンは、キャッシュロックダウン ブロック単位で実行されます。

近似値への丸め (RN) モード
丸め結果が、丸め前の結果に最も近い、表現可能な数値となることを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Memory coherency
Is the problem of ensuring that when a memory location is read (either by a data read or an instruction fetch), the value actually obtained is always the value that was most recently written to the location. This can be difficult when there are multiple possible physical locations, such as main memory, a write buffer and/or cache(s).

Memory Management Unit (MMU)
Provides detailed control of a memory system. Most of the control is provided via translation tables held in memory.

Memory-mapped I/O
Uses special memory addresses that supply I/O functions when they are loaded from or stored to.

Memory Protection Unit (MPU)
Is a hardware unit whose registers provide simple control of a limited number of protection regions in memory.

Mixed-endian
A processor supports mixed-endian memory accesses if accesses to big-endian data and little-endian data can be freely intermixed, with only small performance and code size penalties for doing so.

Modified Virtual Address (MVA)
Is the address produced by the FCSE that is sent to the rest of the memory system to be used in place of the normal virtual address. From ARMv6, use of the FCSE is deprecated, and the FCSE is optional in ARMv7. When the FCSE is absent or disabled the MVA and the *Virtual Address* (VA) have the same value.

MMU *See* Memory Management Unit.

MPU *See* Memory Protection Unit.

Multi-copy atomicity
Is the form of atomicity described in *Multi-copy atomicity* on page A3-28.

See also Atomicity, Single-copy atomicity.

MVA *See* Modified Virtual Address.

NaN NaNs are special floating-point values that can be used when neither a numeric value nor an infinity is appropriate. NaNs can be *quiet* NaNs that propagate through most floating-point operations, or *signaling* NaNs that cause Invalid Operation floating-point exceptions when used. For details, see the IEEE 754 standard.

Observer
A processor or mechanism in the system, such as a peripheral device, that can generate reads from or writes to memory.

Offset addressing
Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

PA *See* Physical address.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

空間的局所性
プログラムがメモリ位置へアクセスした後で、近くのメモリ位置にも直後にアクセスする可能性が高いという考察結果。マルチワード キャッシュラインを持つキャッシュは、この効果を利用してパフォーマンスを向上しています。

クワイエット NaN
ほとんどの浮動小数点演算を通して、変更なしに伝播される NaN。

構成 リセット時またはリセットの直後に行われる設定で、通常はプログラムの実行を通して静的に保持されます。

高速コンテキストスイッチ拡張機能 (FCSE)
ARM メモリシステムの動作を変更し、ARM プロセッサ上で実行されている複数のプログラムが同じアドレス範囲を使用することが許可されます。同時に、メモリシステムの他の場所から見たアドレスは別のものになることが保証されます。ARMv6 から、FCSE の使用は非推奨になっており、ARMv7 では FCSE がオプションです。

コンテキストスイッチ
スレッドやプロセス間で切り替えを行うとき、計算の状態を保存および復元すること。本書では、コンテキストスイッチという用語はオペレーティングシステムによってコンテキストが切り替えられるすべての状況を指し、アドレス空間が変更されるかどうかは関係しません。

サブアーキテクチャ定義
動作がサブアーキテクチャの定義で指定されることを想定していることを意味します。一般に、この定義は複数の実装で共有されますが、特定のタイプのコードのみがこの定義を前提とする必要があります。これによって、新しいサブアーキテクチャの開発が必要なとき、ソフトウェアの変更の必要が最小限になります。

本書では、次の用途にサブアーキテクチャ定義が使用されています。

- VFP 実装とそのサポートコードとのインタフェース
- Jazelle 拡張機能実装とイネーブルド JVM とのインタフェース

時間的局所性
プログラムがメモリ位置へアクセスした後で、近いうちに同じメモリ位置へ再度アクセスする可能性が高いという考察結果。キャッシュはこの効果を利用してパフォーマンスを向上しています。

シグナル NaN
浮動小数点演算のオペランドにシグナル NaN が指定されると、常に無効演算例外が発生します。シグナル NaN はデバッグで、初期化されていない変数の使用の一部を追跡するために使用できます。

自己変更コード
1 つまたは複数の命令をメモリに書き込んでから、それらの命令を実行するコード。自己変更コードを使用するときは、キャッシュ保守およびバリア命令を使用して同期を保証する必要があります。詳細については、*キャッシュの保守操作と分岐予測器の保守操作との順序付け* P. B2-21 を参照して下さい。

実行ストリーム
プログラムのシーケンシャルな実行により実行される命令のストリーム。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Physical address (PA)
Identifies a main memory location.

Post-indexed addressing
Means that the memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

Prefetching
Is the process of fetching instructions from memory before the instructions that precede them have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

Pre-indexed addressing
Means that the memory address is formed in the same way as for offset addressing, but the memory address is also written back to the base register.

Privileged mode
Is any processor mode other than User mode. Memory systems typically check memory accesses from privileged modes against supervisor access permissions rather than the more restrictive user access permissions. The use of some instructions is also restricted to privileged modes.

Process ID
In the FCSE, this is a 7-bit number that identifies which process block the current process is loaded into.

Protection region
Is a memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

Protection Unit
See Memory Protection Unit.

Pseudo-instruction
UAL assembler syntax that assembles to an instruction encoding that is expected to disassemble to a different assembler syntax, and is described in this manual under that other syntax. For example, `MOV <Rd>,<Rm>, LSL #<n>` is a pseudo-instruction that is expected to disassemble as `LSL <Rd>,<Rm>,#<n>`

Quiet NaN
Is a NaN that propagates unchanged through most floating-point operations.

RAO *See* Read-As-One.

RAZ *See* Read-As-Zero.

RAO/SBOP
Read-As-One, Should-Be-One-or-Preserved on writes.

In any implementation, the bit must read as 1, or all 1s for a bit field, and writes to the field must be ignored.

Software can rely on the bit reading as 1, or all 1s for a bit field, but must use an SBOP policy to write to the field.

RAO/WI Read-As-One, Writes Ignored.

In any implementation, the bit must read as 1, or all 1s for a bit field, and writes to the field must be ignored

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

実装定義
動作がアーキテクチャ上で定義されていないために、実装ごとに定義して文書化する必要があることを意味します。

修飾仮想アドレス (MVA)
FCSE により生成されるアドレスで、メモリシステムの他の部分に送られ、通常の仮想アドレスの代わりに使用されます。ARMv6 およびそれ以降では FCSE の使用は非推奨で、ARMv7 では FCSE がオプションです。FCSE が存在しないか、非稼動の場合は、MVA と*仮想アドレス* (VA) は同じ値です。

上位レジスタ
ARM コアレジスタの 8 ～ 15 で、一部の Thumb 命令によりアクセス可能です。

条件付き実行
条件コードフラグが、命令の実行開始時に該当する条件が真であることを示している場合、命令が通常に実行されることを意味します。それ以外の場合、命令は何も実行しません。

条件フィールド
命令が実行可能な条件を指定する、命令内の 4 ビットのフィールド。

シングルコピーのアトミック性
シングルコピーのアトミック性 P. A3- 27 で説明されているアトミック性の形式。

参照先：アトミック性、マルチコピーのアトミック性。

セキュリティホール
システムの保護をバイパスする機構。

セットアソシエティビティ
キャッシュの各キャッシュセットに含まれている、キャッシュラインの数。≥ 1 の任意の数で、2 のべき乗に制限する必要はありません。

ダイレクト メモリアクセス
プロセッサによるデータへのアクセスを行わず、メインメモリへ直接アクセスする操作。

タグビット

仮想アドレスのビット [31:L + S]。ここで $L = \log_2$（キャッシュラインの長さ）、$S = \log_2$（キャッシュセットの数）です。ARM プロセッサから供給された仮想アドレスのタグビットが、選択されたキャッシュセットで有効なラインに関連付けられているタグビットと一致するとき、キャッシュヒットが発生します。

ダブルワード
64 ビットのデータ項目。ARM システムでのダブルワードは、通常は最低でもワードアラインドです。

ダブルワードアラインド
アドレスが 8 の倍数であることを意味します。

単精度値
32 ビットワードで、メモリに保持するときはワードアラインドの必要があります。IEEE 754-1985 規格に従った基本単精度浮動小数点数値として解釈されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Software can rely on the bit reading as 1, or all 1s for a bit field, and on writes being ignored.

RAZ/SBZP
Read-As-Zero, Should-Be-Zero-or-Preserved on writes.

In any implementation, the bit must read as 0, or all 0s for a bit field, and writes to the field must be ignored.

Software can rely on the bit reading as 0, or all 0s for a bit field, but must use an SBZP policy to write to the field.

RAZ/WI Read-As-Zero, Writes Ignored.

In any implementation, the bit must read as 0, or all 0s for a bit field, and writes to the field must be ignored.

Software can rely on the bit reading as 0, or all 0s for a bit field, and on writes being ignored.

Read-allocate cache
Is a cache in which a cache miss on reading data causes a cache line to be allocated into the cache.

Read-As-One (RAO)
In any implementation, the bit must read as 1, or all 1s for a bit field.

Read-As-Zero (RAZ)
In any implementation, the bit must read as 0, or all 0s for a bit field.

Read, modify, write
In a read, modify, write instruction sequence, a value is read to a general-purpose register, the relevant fields updated in that register, and the new value written back.

Reserved
Unless otherwise stated:

- instructions that are reserved or that access reserved registers have UNPREDICTABLE behavior
- bit positions described as Reserved are UNK/SBZP.

RISC Reduced Instruction Set Computer.

Rounding error
Is defined to be the value of the rounded result of an arithmetic operation minus the exact result of the operation.

Rounding modes
Specify how the exact result of a floating-point operation is rounded to a value that is representable in the destination format.

Round to Nearest (RN) mode
Means that the rounded result is the nearest representable number to the unrounded result.

Round towards Plus Infinity (RP) mode
Means that the rounded result is the nearest representable number that is greater than or equal to the exact result.

Round towards Minus Infinity (RM) mode
Means that the rounded result is the nearest representable number that is less than or equal to the exact result.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

直接マップキャッシュ
1 ウェイのセットアソシエイティブキャッシュ。各キャッシュセットは、単一のキャッシュラインで構成されているため、キャッシュのルックアップでは、単一のキャッシュラインが選択されてチェックが行なわれます。

常に 0(SBZ)
ソフトウェアで 0（ビットフィールドの場合はすべてのビットが 0）を書き込む必要があります。0 以外の値を書き込んだ場合の結果は予測不能です。

常に 0 または保持 (SBZP)
以前に読み出しを行わずに値を書き込む場合、またはレジスタが初期化されていない場合は、ソフトウェアで 0（ビットフィールドの場合はすべてのビットが 0）を書き込む必要があります。そのプロセッサが最後にリセットされた後で同じプロセッサからそのレジスタを読み出したことがある場合は、以前に読み出した値を書き込むことでフィールドの値を保持する必要があります。

ハードウェアは、これらのフィールドへの書き込みを無視する必要があります。

0（ビットフィールドの場合はすべてのビットが 0）でもなく、同じプロセッサの同じフィールドから以前に読み出した値でもない値をこのフィールドへ書き込んだ場合、結果は予測不能です。

常に 1(SBO)
ソフトウェアで 1（ビットフィールドの場合はすべてのビットが 1）を書き込む必要があります。1 以外の値を書き込んだ場合の結果は予測不能です。

常に 1 または保持 (SBOP)
以前に読み出しを行わずに値を書き込む場合、またはレジスタが初期化されていない場合は、ソフトウェアで 1（ビットフィールドの場合はすべてのビットが 1）を書き込む必要があります。そのプロセッサが最後にリセットされた後で同じプロセッサからそのレジスタを読み出したことがある場合は、以前に読み出した値を書き込むことでフィールドの値を保持する必要があります。

ハードウェアは、これらのフィールドへの書き込みを無視する必要があります。

1（ビットフィールドの場合はすべてのビットが 1）でもなく、同じプロセッサの同じフィールドから以前に読み出した値でもない値をこのフィールドへ書き込んだ場合、結果は予測不能です。

データキャッシュ
データのロード / ストアを処理する目的にのみ使用される、独立したキャッシュ。

デジタル信号処理 (DSP)
サンプリングされてデジタル形式に変換された信号を処理するために使用される各種のアルゴリズムを指します。このようなアルゴリズムでは、飽和演算が多く使用されます。

統一キャッシュ
命令フェッチとデータのロード / ストア、両方の処理に使用されるキャッシュ。

特権モード
ユーザモード以外のすべてのプロセッサモード。メモリシステムは通常、特権モードからのメモリアクセスを、より制限の多いユーザアクセスではなく、スーパーバイザアクセス許可に対してチェックします。また一部の命令は、特権モードでのみ使用できます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Round towards Zero (RZ) mode
Means that results are rounded to the nearest representable number that is no greater in magnitude than the unrounded result.

Saturated arithmetic
Is integer arithmetic in which a result that would be greater than the largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in ARM processors, in which overflowing results wrap around from $+2^{31}-1$ to -2^{31} or vice versa.

SBO *See* Should-Be-One.

SBOP *See* Should-Be-One-or-Preserved.

SBZ *See* Should-Be-Zero.

SBZP *See* Should-Be-Zero-or-Preserved.

Security hole
Is a mechanism that bypasses system protection.

Self-modifying code
Is code that writes one or more instructions to memory and then executes them. When using self-modifying code you must use cache maintenance and barrier instructions to ensure synchronization. For details see *Ordering of cache and branch predictor maintenance operations* on page B2-21.

Set-associativity
Is the number of cache lines in each of the cache sets in a cache. It can be any number ≥ 1, and is not restricted to being a power of two.

Should-Be-One (SBO)
Should be written as 1, or all 1s for a bit field, by software. Values other than 1 produce UNPREDICTABLE results.

Should-Be-One-or-Preserved (SBOP)
Must be written as 1, or all 1s for a bit field, by software if the value is being written without having been previously read, or if the register has not been initialized. Where the register was previously read on the same processor, since the processor was last reset, the value in the field should be preserved by writing the value that was previously read.

Hardware must ignore writes to these fields.

If a value is written to the field that is neither 1 (or all 1s for a bit field), nor a value previously read for the same field on the same processor, the result is UNPREDICTABLE.

Should-Be-Zero (SBZ)
Should be written as 0, or all 0s for a bit field, by software. Values other than 0 produce UNPREDICTABLE results.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ドメイン メモリのセクション、ラージページ、スモールページの集まりで、独自のアクセス許可を設定できます。ドメインに設定されたアクセス許可は、CP15 c3 のドメインアクセス制御レジスタに書き込むことで、迅速に切り替えることができます。

トラップイネーブル ビット
トラップまたはトラップなしのどちらの例外処理を選択するかを決定します。トラップ例外処理が選択された場合、処理方法は実装定義です。

ハーフワード
16 ビットのデータ項目。ARM システムでのハーフワードは、通常はハーフワードアラインドです。

ハーフワードアラインド
アドレスが 2 の倍数であることを意味します。

倍精度数値
2 つの 32 ビットワードで構成されます。これら 2 つのワードはメモリ上で連続していて、いずれもワードアラインしている必要があり、IEEE 754-1985 規格に従って基本倍精度浮動小数点数として解釈されます。

バイト 8 ビットのデータ項目。

ハイベクタ
例外ベクタの代替位置。ハイベクタのアドレス範囲は、アドレス空間の下位ではなく、最上位付近にあります。

バンクレジスタ
複数のインスタンスを持つレジスタで、使用されるインスタンスはプロセッサのモード、セキュリティ状態、その他プロセッサの状態に応じて変化します。

汎用レジスタ
32 ビット汎用整数レジスタ R0 ～ R15 のうち 1 つ。R15 はプログラムカウンタを保持しているため、R0 ～ R14 には適用されない多くの使用制限があることに注意して下さい。

ビッグエンディアン メモリ
次の条件を満たすメモリ。
- ワードアラインされたアドレスのバイトまたはハーフワードが、そのアドレスのワード内で最上位のバイトまたはハーフワードである。
- ハーフワードアラインされたアドレスのバイトが、そのアドレスのハーフワード内で最上位バイトである。

フォールト
MMU により生成されるアボート。

符号付きイミディエートおよびオフセットフィールド
特記されていない限り、2 の補数表記でエンコードされます。

符号付きデータタイプ
-2^{N-1} ～ $+2^{N-1}-1$ の範囲で、2 の補数形式を使用して表現される整数。

符号なしデータタイプ
0 ～ $+2^{N}-1$ の範囲の負でない整数で、通常のバイナリ形式を使用します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Should-Be-Zero-or-Preserved (SBZP)
Must be written as 0, or all 0s for a bit field, by software if the value is being written without having been previously read, or if the register has not been initialized. Where the register was previously read on the same processor, since the processor was last reset, the value in the field should be preserved by writing the value that was previously read.

Hardware must ignore writes to these fields.

If a value is written to the field that is neither 0 (or all 0s for a bit field), nor a value previously read for the same field on the same processor, the result is UNPREDICTABLE.

Signaling NaNs
Cause an Invalid Operation exception whenever any floating-point operation receives a signaling NaN as an operand. Signaling Nans can be used in debugging, to track down some uses of uninitialized variables.

Signed data types
Represent an integer in the range -2^{N-1} to $+2^{N-1}-1$, using two's complement format.

Signed immediate and offset fields
Are encoded in two's complement notation unless otherwise stated.

SIMD Means Single-Instruction, Multiple-Data operations.

Single-copy atomicity
Is the form of atomicity described in *Single-copy atomicity* on page A3-27.

See also Atomicity, Multi-copy atomicity.

Single-precision value
Is a 32-bit word, that must be word-aligned when held in memory, and that is interpreted as a basic single-precision floating-point number according to the IEEE 754-1985 standard.

Spatial locality
Is the observed effect that after a program has accessed a memory location, it is likely to also access nearby memory locations in the near future. Caches with multi-word cache lines exploit this effect to improve performance.

SUBARCHITECTURE DEFINED
Means that the behavior is expected to be specified by a subarchitecture definition. Typically, this will be shared by multiple implementations, but it must only be relied on by specified types of code. This minimizes the software changes required when a new subarchitecture has to be developed.

In this manual, subarchitecture definitions are used for:

- the interface between a VFP implementation and its support code
- the interface between an implementation of the Jazelle extension and an Enabled JVM.

Tag bits Are bits [31:L+S]) of a virtual address, where L = $\log_2$(cache line length) and S = $\log_2$(number of cache sets). A cache hit occurs if the tag bits of the virtual address supplied by the ARM processor match the tag bits associated with a valid line in the selected cache set.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

物理アドレス (PA)
メインメモリ上の位置を示します。

プラス無限大への丸め (RP) モード
丸め結果が、正確な結果と等しいかより大きい、最も近い表現可能な数値となることを意味します。

フラット アドレスマッピング
すべてのアクセスの物理アドレスが仮想アドレスと同一であるマッピング。

プリインデクス アドレシング
メモリアドレスをオフセットアドレシングと同じ方法で計算するが、そのメモリアドレスをベースレジスタへ書き戻すことを意味します。

プリフェッチ
前の命令の実行が完了する前に、メモリから命令をフェッチする処理。プリフェッチされた命令は、必ず実行されるとは限りません。

ブレークポイント
特定の命令の実行によりトリガされるデバッグイベントで、命令のアドレス、命令が実行されたときのプロセッサの状態、またはその両方によって指定されます。

プロセス ID
FCSR において、現在のプロセスがロードされるプロセスブロックを識別する 7 ビットの番号。

ブロッキング
操作が完了するまで、以後の命令の実行を許可しないような操作を指します。

非ブロッキングの操作では、操作が完了する前に次の命令を実行することが許可され、例外が発生した場合にはプロセッサに例外が通知されません。これにより、正確なプロセッサ状態を保持しなくても、非ブロッキング操作を実行している間に後続の命令を完了するような実装が可能です。

分岐予測
プロセッサが、その先プリフェッチする実行パスを選択すること（プリフェッチ参照）。例えば、分岐命令の後でプロセッサは分岐に続く命令かまたは分岐先の命令のどちらかをプリフェッチできます。

ベースレジスタ
命令のアドレス計算の基準値として、ロード / ストア命令で指定されるレジスタ。命令とそのアドレシングモードによっては、メモリに送られる仮想アドレスを形成するために、ベースアドレスの値にオフセットを加算または減算することができます。

ベースレジスタ ライトバック
アドレス計算時に、修飾された値をベースレジスタへ書き戻すことを指します。

変換テーブル
メモリに保持されるテーブル。1KB から 1MB までの各サイズのメモリ領域について、プロパティを定義します。

変換テーブルウォーク
完全な変換テーブルルックアップを実行する処理。ハードウェアによって自動的に実行されます。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Temporal locality
Is the observed effect that after a program has accesses a memory location, it is likely to access the same memory location again in the near future. Caches exploit this effect to improve performance.

Thumb instruction
Is one or two halfwords that specify an operation for a processor in Thumb state to perform. Thumb instructions must be halfword-aligned.

TLB *See* Translation Lookaside Buffer.

TLB lockdown
Is a way to prevent specific translation table walk results being accessed. This ensures that accesses to the associated memory areas never cause a translation table walk.

Translation Lookaside Buffer (TLB)
Is a memory structure containing the results of translation table walks. They help to reduce the average cost of a memory access. Usually, there is a TLB for each memory interface of the ARM implementation.

Translation tables
Are tables held in memory. They define the properties of memory areas of various sizes from 1KB to 1MB.

Translation table walk
Is the process of doing a full translation table lookup. It is performed automatically by hardware.

Trap enable bits
Determine whether trapped or untrapped exception handling is selected. If trapped exception handling is selected, the way it is carried out is IMPLEMENTATION DEFINED.

Unaligned
An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

Unaligned memory accesses
Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

Unallocated
Except where otherwise stated, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as UNDEFINED, UNPREDICTABLE, or an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

UNDEFINED
Indicates an instruction that generates an Undefined Instruction exception.

See also Undefined Instruction exception on page B1-49.

Unified cache
Is a cache used for both processing instruction fetches and processing data loads and stores.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

変換ルックアサイドバッファ（ＴＬＢ）
変換テーブルウォークの結果を含むメモリ構造。メモリアクセスの平均コストを減らすために役立ちます。通常は、ARM 実装の各メモリインタフェースに TLB が存在します。

変更不可 (DNM)
ソフトウェアで値を変更できないことを意味します。DNM フィールドは、未知の値として読み出されるため、同じプロセッサの同じフィールドから読み出された同じ値だけを書き込む必要があります。

飽和演算
表現可能な最大の数値より大きな結果を表現可能な最大の数値に設定し、表現可能な最小の数値より小さな結果を表現可能な最小の数値に設定する整数演算。符号付きの飽和演算は、DPS アルゴリズムで多く使用されます。これに対して、ARM プロセッサで使用される通常の符号付き整数演算では、オーバフローした結果は $+2^{31}-1$ 〜 -2^{31} へ、またはその逆にラップアラウンドされます。

保護ユニット
参照先：メモリ保護ユニット。

保護領域
メモリ保護ユニットのレジスタによって位置、サイズ、その他のプロパティが定義されているメモリ領域。

ポストインデクス アドレシング
メモリアドレスとしてベースレジスタの値を使用しますが、ベースレジスタの値にオフセットを加算または減算して、結果をベースレジスタへ書き戻すことを意味します。

マイナス無限大への丸め (RM) モード
丸め結果が、正確な結果と等しいかより小さい、最も近い表現可能な数値となることを意味します。

マルチコピーのアトミック性
マルチコピーのアトミック性 P. A3-28 で説明されているアトミック性の形式。

参照先：アトミック性、シングルコピーのアトミック性。

丸めエラー
数値演算の丸め結果と、演算の正確な結果との差と定義されます。

丸めモード
浮動小数点演算の正確な結果を、デスティネーションフォーマットで表現可能な値に丸める方式を指定します。

未知
未知の値には有効なデータが含まれておらず、時間ごと、命令ごと、実装ごとに変化する可能性があります。未知の値がセキュリティホールとなることは回避する必要があります。未知な値を文書化したり、定義された値や効果を持つものとして取り扱ったりしてはいけません。

未定義 未定義命令例外を生成する命令を指します。

参照先：*未定義命令例外* P. B1-49。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Unindexed addressing
Means addressing in which the base register value is used directly as the virtual address to send to memory, without adding or subtracting an offset. In most types of load/store instruction, unindexed addressing is performed by using offset addressing with an immediate offset of 0. The LDC, LDC2, STC, and STC2 instructions have an explicit unindexed addressing mode that permits the offset field in the instruction to be used to specify additional coprocessor options.

UNKNOWN
An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not be a security hole. UNKNOWN values must not be documented or promoted as having a defined value or effect.

UNK/SBOP
UNKNOWN on reads, Should-Be-One-or-Preserved on writes.

In any implementation, the bit must read as 1, or all 1s for a bit field, and writes to the field must be ignored.

Software must not rely on the bit reading as 1, or all 1s for a bit field, and must use an SBOP policy to write to the field.

UNK/SBZP
UNKNOWN on reads, Should-Be-Zero-or-Preserved on writes.

In any implementation, the bit must read as 0, or all 0s for a bit field, and writes to the field must be ignored.

Software must not rely on the bit reading as 0, or all 0s for a bit field, and must use an SBZP policy to write to the field.

UNK Is an abbreviation indicating that software must treat a field as containing an UNKNOWN value.

In any implementation, the bit must read as 0, or all 0s for a bit field. Software must not rely on the field reading as zero.

UNPREDICTABLE
Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not represent security holes. UNPREDICTABLE behavior must not halt or hang the processor, or any parts of the system. UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

Unsigned data types
Represent a non-negative integer in the range 0 to $+2^N-1$, using normal binary format.

VA *See* Virtual address.

VFP Is a coprocessor extension to the ARM architecture. It provides single-precision and double-precision floating-point arithmetic.

Virtual address (VA)
Is an address generated by an ARM processor. For a PMSA implementation, the virtual address is identical to the physical address.

Watchpoint
Is a debug event triggered by an access to memory, specified in terms of the address of the location in memory being accessed.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

未割り当て
特記されていない限り、アーキテクチャが命令のビットパターン全体に特定の機能を割り当てておらず、代わりに未定義、予測不能、または未割り当てのヒント命令と記述されている場合、その命令エンコードは未割り当てです。

レジスタのビットについては、アーキテクチャでそのビットに機能が割り当てられていない場合、そのビットは未割り当てです。

明示的アクセス
プロセッサで実行されるロード / ストア命令により生成されるメモリからの読み出し、またはメモリへの書き込み。L1 DMA アクセスや、ハードウェア変換テーブルアクセスにより生成される読み出しや書き込みは、明示的アクセスではありません。

命令キャッシュ
命令フェッチの処理にのみ使用される、独立したキャッシュ。

メモリコヒーレンシ
メモリ位置が読み出された（データ読み出しまたは命令フェッチにより）とき、実際に読み出される値が、その位置に最後に書き込まれた値と常に同じであることを保証する問題。物理的なメモリがメインメモリ、ライトバッファ、キャッシュなど複数の場所に分かれている場合、この問題の解決は簡単ではないことがあります。

メモリバリア
メモリバリア P. A3- 47 参照。

メモリマップド I/O
I/O 機能に対してロードやストアを行うとき、特別なメモリアドレスを使用する I/O 方式。

メモリ管理ユニット (MMU)
メモリシステムの細かい制御を行います。ほとんどの制御は、メモリに保持されている変換テーブルを使用して行われます。

メモリ保護ユニット (MPU)
レジスタを使用してメモリ内に限られた数の保護領域を設定し、単純な制御を行うハードウェアユニット。

予測不能
動作を予測できないことを意味します。予測不能な動作がセキュリティホールとならないようにする必要があります。予測不能な動作によって、プロセッサまたはシステムのどの部分も停止やハングアップすることは許されません。予測不能な動作を文書化したり、定義された効果を持つものとして取り扱ったりしてはいけません。

呼び出し先保存レジスタ
呼び出されたプロシージャが保存する必要のあるレジスタ。呼び出し先保存レジスタを保存するため、呼び出されたプロシージャは一般にそのレジスタを一切使用しないか、プロシージャの開始時にレジスタをスタックに保存して、プロシージャの終了時に再ロードします。

呼び出し元保存レジスタ
呼び出されたプロシージャが保存する必要のないレジスタ。呼び出し元のプロシージャが値を保持する必要がある場合、呼び出し元で保存と再ロードを行う必要があります。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

Word Is a 32-bit data item. Words are normally word-aligned in ARM systems.

Word-aligned
Means that the address is divisible by 4.

Write-Allocate cache
Is a cache in which a cache miss on storing data causes a cache line to be allocated into the cache.

Write-Back cache
Is a cache in which when a cache hit occurs on a store access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in main memory. Any such data is written back to main memory when the cache line is cleaned or re-allocated. Another common term for a Write-Back cache is a *copy-back cache*.

Write-Through cache
Is a cache in which when a cache hit occurs on a store access, the data is written both to the cache and to main memory. This is normally done via a write buffer, to avoid slowing down the processor.

Write buffer
Is a block of high-speed memory whose purpose is to optimize stores to main memory.

 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

読み出し値 0(RAZ)
このビットは、すべての実装で 0（ビットフィールドの場合はすべてのビットが 0）として読み出される必要があります。

読み出し値 1(RAO)
このビットは、すべての実装で 1（ビットフィールドの場合はすべてのビットが 1）として読み出される必要があります。

読み出し－変更－書き込み
読み出し－変更－書き込みの命令シーケンスでは、汎用レジスタに値が読み出され、目的のフィールドが更新されてから、新しい値が書き戻されます。

読み出し割り当てキャッシュ
データの読み出し時のキャッシュミスにより、キャッシュ内のキャッシュラインが割り当てられる方式のキャッシュ。

予約 特に記載がない限り、次の規則が適用されます。
- 予約されている命令、または予約されているレジスタへのアクセスの動作は予測不能です。
- 予約と記載されているビット位置は UNK/SBZP です。

ライトスルー キャッシュ
この方式のキャッシュでは、ストアアクセスでキャッシュヒットが発生した場合、データはキャッシュとメインメモリの両方に書き込まれます。プロセッサの速度低下を避けるため、この動作は通常はライトバッファ経由で行われます。

ライトバック キャッシュ
この方式のキャッシュでは、ストアアクセスでキャッシュヒットが発生した場合、データはキャッシュにのみ書き込まれます。このため、キャッシュのデータはメインメモリのデータより新しいことになります。このようなデータは、キャッシュラインのクリーニングまたは再割り当てのときにメインメモリへ書き戻されます。ライトバック キャッシュを指す別の一般的な用語として、*コピーバック キャッシュ*があります。

ライトバッファ
高速なメモリのブロックで、メインメモリへのストアを最適化することを目的としています。

リトルエンディアン メモリ
次の条件を満たすメモリ。
- ワードアラインされたアドレスのバイトまたはハーフワードが、そのアドレスにあるワード内の最下位バイトまたはハーフワードである。
- ハーフワードアラインされたアドレスのバイトが、そのアドレスのハーフワード内の最下位バイトである。

例外 イベントを処理します。例えば、外部割り込みや未定義命令を処理できます。

例外ベクタ
下位メモリにある多くの固定アドレスの 1 つ。ハイベクタが構成されている場合は上位メモリにあります。

例外モード
特定の例外が発生したときに開始される特権モード。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。
 Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.

抄訳

ロード / ストアアーキテクチャ
データ処理動作が、直接メモリの内容に対して行われるのではなく、レジスタの内容に対してのみ行われるアーキテクチャ。

ロング分岐
ロード命令を使用して、4GB のアドレス空間内で任意の場所へ分岐すること。

ワード　32 ビットのデータ項目。ARM システムでのワードは、通常はワードアラインドです。

ワードアラインド
アドレスが 4 の倍数であることを意味します。

ここに記載される内容は英語原文より抄訳されたものです。抄訳が原文に相違する場合はすべて原文が優先します。原文は左のページよりお読みいただけます。

Copyright © 1996-1998, 2000, 2004-2008 ARM Limited. All rights reserved.