ARMv7-M アーキテクチャ アプリケーションレベル リファレンス マニュアル

ARM®

ARM DDI 0405B

ARMv7-M アーキテクチャ アプリケーションレベル リファレンスマニュアル

リリース情報
本書には、以下の変更が加えられています。

改訂履歴

日付	変更箇所	公開の有無	変更内容
2006 年 3 月	A-01	公開	ベータリリース（www.arm.com からダウンロード可能）
2007 年 8 月	B	公開	初版

著作権表記

110 Fulbourn Road Cambridge, England CB1 9NJ

目次
ARMv7-M アーキテクチャ アプリケーション レベル リファレンスマニュアル

表目次
ARMv7-M アーキテクチャ アプリケーション レベル リファレンスマニュアル

図目次
ARMv7-M アーキテクチャ アプリケーション レベル リファレンスマニュアル

序章

本章では、本書の内容について説明し、使用する表記規則と用語のリストを示します。

本書について

本書は、ARM アーキテクチャバージョン 7(ARMv7-M) に関連するマイクロコントローラプロファイルの簡略版ドキュメントです。すべての ARMv7 プロファイルの簡潔な定義については、P.A1-1 を参照して下さい。

本書は、次の 3 つのパートから構成されています。

パート A　命令セットを含む、アプリケーションプログラマから参照可能なアプリケーションレベルのプログラミングモデルとメモリモデルの情報

これは、アプリケーションのプログラムや、デバッガを除くツールチェーンコンポーネント（コンパイラ、リンカ、アセンブラ、逆アセンブラ）の開発に必要な情報です。ARMv7-M では、これらのほぼ全体が、他の 2 つのプロファイルと共通の資料のサブセットです。プロファイル間で異なる命令セットの詳細については、明確に説明されています。

注

すべての ARMv7 プロファイルは、共通のプロシージャコール規約である ARM アーキテクチャプロシージャ呼び出し標準 (AAPCS) をサポートしています。

パート B　システムレベルのサポート命令を含む、システムの正しさのために必要とされるシステムレベルのプログラミングモデルの概要について説明します。パート B は、デバイスのテクニカル リファレンスマニュアル(TRM) と同時に読むことを意図して作成されています。

システムレベルでは、ARMv7-M 例外モデルがサポートされています。さらに、プロセッサリソースの構成と制御、およびメモリアクセス権の管理機能も用意されています。オペレーティングシステム (OS) の移植やシステムサポートソフトウェアの開発には、TRM に記載されている関連事項の詳細を知る必要があります。

このパートはプロファイル固有です。ARMv7-Mは新しいプログラマモデルを導入しているため、システムレベルで一部他のプロファイルと根本的に異なる点があります。ARMv7-M はメモリマップドアーキテクチャであるため、システム メモリマップについてもここで解説します。

パート C　ARMv7-M デバッグ機能の概要について説明します。パート C は、デバイスのテクニカル リファレンスマニュアル (TRM) に記載されているデバッグの詳細、および ARMv7-M デバッグツール ユーザマニュアルと同時に読むことを意図して作成されています。

ARMv7-M では、次のタイプのデバッグがサポートされています。

- ホールトデバッグおよび関連するデバッグ状態
- 例外ベースのモニタデバッグ
- 外部エージェントに対するイベント生成および送信のための非侵襲的なサポート

このパートはプロファイル固有であり、ARMv7 アーキテクチャ内でマイクロコントローラプロファイルに固有のデバッグ機能がいくつか含まれています。

本書の使用法

本書の情報は、次に説明する章およびそれをサポートする一連の付録で構成されています。

P.A1-1「*はじめに*」

ARMv7 の概要、各種のアーキテクチャプロファイル、およびマイクロコントローラ (M) プロファイルの背景について説明します。

P.A2-1「*アプリケーションレベルのプログラマモデル*」

アプリケーションレベルで使用可能なレジスタとステータスビットの詳細、および例外のサポートの概要について説明します。

P.A3-1「*ARM アーキテクチャのメモリモデル*」

ARM アーキテクチャのメモリ属性およびメモリオーダモデルの詳細について説明します。

P.A4-1「*ARMv7-M 命令セット*」

Thumb 命令セットの一般的な情報について説明します。

P.A5-1「*Thumb®*」

Thumb 命令セットのエンコード図を示し、ビットフィールドの使用法、未定義および予測不能という用語について説明します。

P.A6-1「*Thumb 命令の詳細*」

各 Thumb 命令に関する詳細な参照資料で、命令ニーモニックのアルファベット順に解説されています。システム命令の概要の情報は、パート B の詳細な定義に含まれており、この章からも参照されています。

P.B1-1「*システムレベル プログラマモデル*」

システムレベルで使用可能なレジスタ、ステータス、制御機構の詳細について説明します。

P.B2-1「*システム アドレスマップ*」

システム アドレスマップの概要、および専用ペリフェラルバス領域内でアーキテクチャにより定義されている機能の概要について説明します。本章には、保護メモリシステムに対するメモリマップされたサポートの概要も含まれています。

P.B3-1「*ARMv7-M システム命令*」

システムレベルの命令に関する詳細な参照資料です。

P.C1-1「*ARMv7-M デバッグ*」

ARMv7-M デバッグのサポートについて説明します。

P.AppxA-1「*CPUID*」

ARM アーキテクチャ機能の特定に使用される ID 属性レジスタの概要について説明します。

P.AppxB-1「*過去の命令のニーモニック*」

ARM アーキテクチャの以前のバージョンで使用されている Thumb フォーマットについて、命令構文の統一アセンブラ言語形式との相互参照を示します。

P.AppxC-1「*非推奨の ARMv7-M 機能*」

将来の互換性のため、避けることが望ましい非推奨のソフトウェア機能について説明します。ARM 社は、ARM アーキテクチャの将来のバージョンでこの機能を削除する予定です。

P.AppxD-1「*擬似コードの定義*」

メモリモデルおよび命令の操作を説明するための擬似コードによって使用されている用語、フォーマット、ヘルパ関数の定義について説明します。

用語集　専門用語の用語集です。擬似コードに関連する用語は含まれていません。

表記規則

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

書体の一般的な規則

`typewriter` アセンブラ構文の表記、命令の擬似コードの表記、ソースコードの例に使用されています。アセンブラ構文の表記に使用されている表記規則の詳細については、P.A6-3「*アセンブラ構文*」を参照して下さい。擬似コードの表記規則の詳細については、P.AppxD-1「*擬似コードの定義*」を参照して下さい。

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

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

太字 必要に応じて、説明のリストやその他の場所で強調のため使用されます。

小さい大文字 技術的に特別な意味があるいくつかの用語に使用されます。

参考資料

このセクションでは、ARM ファミリのプロセッサに関する追加情報が記載されている出版物を紹介します。本書は、アーキテクチャの情報を解説するドキュメントで、対象となる実装のテクニカル リファレンスマニュアル (TRM) と同時に読むことを意図して作成されています。TRM では、ARM 準拠のコアに含まれる実装定義のアーキテクチャ機能の詳細について説明されています。追加のシステムの詳細については、シリコンパートナーのデバイス仕様を使用する必要があります。

ARM 社は、自社の出版物の定期的な更新と修正を行っています。最新情報および正誤表について、一部の資料は http://www.arm.com で参照できます。または、最近出版された ARM の情報および対象となるデバイス固有の情報のご利用に関しては、販売代理店またはシリコンパートナーにお問い合わせ下さい。最近出版された ARM の情報は、お近くの ARM オフィスにご連絡下さい。

ARM の刊行物

最初の ARMv7-M 実装については、*Cortex-M3 テクニカル リファレンスマニュアル* (ARM DDI 0337) を参照して下さい。

ご意見・ご質問

ARM 社では、本書に関するご意見・ご質問をお待ちしております。

本書に関するご意見

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

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

補足または改善すべき点についてのご提案もお待ちしております。

パート A

アプリケーションレベルのアーキテクチャ

第 A1 章
はじめに

ARMv7 はアーキテクチャプロファイル群のセットとして記述されています。ARMv7 には、次の 3 つのプロファイルが定義されています。

ARMv7-A ARM および Thumb 命令セットをサポートし、メモリ管理モデル内で仮想アドレスのサポートを必要とするシステム向けの「アプリケーションプロファイル」

ARMv7-R ARM および Thumb 命令セットをサポートし、メモリ管理モデル内で物理アドレスのみのサポートを必要とするシステム向けの「リアルタイムプロファイル」

ARMv7-M Thumb 命令セットのみをサポートし、ある実装に対して、全体のサイズと予測可能な応答動作が絶対性能よりも重要なシステム向けの「マイクロコントローラプロファイル」

プロファイルは ARMv7 の開発で正式に導入された概念ですが、A- プロファイルおよび R- プロファイルは以前のバージョンでも暗黙的に存在しており、それぞれ仮想メモリシステム アーキテクチャ (VMSA) および保護メモリシステム アーキテクチャ (PMSA) に対応しています。

命令セットアーキテクチャ (ISA)

ARMv7-M は Thumb 命令のみをサポートしており、Thumb-2 に準拠しています。Thumb-2 は Thumb 実行状態での 32 ビット命令実行のサポートとして定義されています。サポートされている命令の詳細なリストについては、第 A6 章 *「Thumb 命令の詳細」*を参照して下さい。

A1.1 ARM アーキテクチャ : M- プロファイル

ARM アーキテクチャはいくつかの主要な改訂を経て進化し、広範な性能範囲の製品への実装をサポートすることが可能になり、今日では 1 年に 10 億個を超える部品が製造されています。最新のバージョン (ARMv7) では、各種のアーキテクチャプロファイルがバリエーションとして正式に組み込まれ、市場ごとに異なる要求に応じてアーキテクチャをカスタマイズすることが可能になっています。重要な点は、すべてのプロファイルはアプリケーションレベルでは一貫しており、相違点のほとんどはシステムレベルのものであるということです。

ARMv6T2 で導入された Thumb-2 は、ARM 命令セットと Thumb 命令セットの両方の長所を兼ね備えており、新規市場、特にマイクロコントローラ市場で ARM アーキテクチャを活用する可能性が広がりました。この可能性を最大限に利用するため、すでに ARM が優位性を確立している高パフォーマンス市場およびリアルタイムのエンベデッド市場以外の分野を対象とした、Thumb のみを使用する新しいプログラマモデル（ひとつのシステムレベルの見せ方）が固有のプロファイルとして導入されました。

ARMv7-M 実装の主要な特徴は、次のとおりです。

- 消費電力、パフォーマンス、実装面積において、業界で最も優れた実装を実現
  - 単純なパイプライン設計により、広範な市場や応用分野で最先端のシステムパフォーマンスを実現可能
- 確実に予測可能な動作
  - 単一または少ないサイクルでの実行
  - 最小の割り込みレイテンシ（短いパイプライン）
  - キャッシュなし動作
- C/C++ への優れた対応—この分野における ARM のプログラミング規約に対応
  - 例外ハンドラは標準 C/C++ 関数であり、標準呼び出し規則を使用してハンドラに入る
- エンベデッドシステムへの対応を念頭に置いた設計
  - 少ないピン数のデバイス
  - ARM アーキテクチャのエントリレベル市場への新規参入を可能に
- イベント駆動型システムのデバッグおよびソフトウェアプロファイリングのサポート

本書で解説する内容は、ARMv7-M プロファイルに固有のものです。

第 A2 章
アプリケーションレベルのプログラマモデル

本章では、アプリケーションレベルから見たプログラマモデルについて説明します。これはアプリケーション開発に必要な情報であり、オペレーティングシステムのもとでのサービスおよびサポートアプリケーションの実行に必要なシステム情報とは明確に区別されています。本章は以下のセクションから構成されています。

システム関連情報は、必要に応じて概要の形式や、アーキテクチャ仕様のシステム情報の部分に関する参照として提供されます。

A2.1 アプリケーションレベルのプログラマモデルについて

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

本章の情報は、オペレーティングシステムのもとでのサービスおよびサポートアプリケーションの実行に必要なシステム情報とは明確に区別されています。システム情報の概要については、第 B1 章 *「システム レベル プログラマモデル」* を参照して下さい。

A2.1.1 特権実行

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

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

A2.1.2 システムレベル アーキテクチャ

スレッドモードは、ARMv7-M でのアプリケーション実行における基本的なモードで、リセット時に選択されます。スレッドモードでは、SVC 命令を使用してスーパバイザコールを生成するか、またはシステムアクセスおよび制御を直接処理することができます。

すべての例外はハンドラモードで実行されます。スーパバイザコール (SVCall) ハンドラは、周辺装置とのやり取り、メモリ割り当て、ソフトウェアスタックの管理などのリソース管理作業を、アプリケーションの代わりに実行します。

本章では、システムレベルの情報については必要に応じて、次の内容にのみ言及します。

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

A2.2 ARM コアのデータタイプおよび演算

ARMv7-M プロセッサは、メモリ上で次のデータタイプをサポートしています。

バイト 8 ビット
ハーフワード 16 ビット
ワード 32 ビット

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

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

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

命令セットには、2 つ以上のワードをメモリとの間でロード / ストアする操作が含まれています。64 ビット整数のロード / ストアには、これらの命令を使用します。

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

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

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

A2.2.1 整数演算

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

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

シフトおよびローテート演算

命令内では、次のタイプのシフトおよびローテート演算が使用されます。

論理左シフト (LSL)

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

論理右シフト (LSR)

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

算術右シフト (ASR)

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

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

拡張付き右ローテート (RRX)

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

シフトおよびローテート演算の擬似コードの詳細

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

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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



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

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

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

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

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

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

加算および減算の擬似コードの詳細

擬似コードにおいては、加算および減算はビット数の制約のない整数およびビットストリングのいずれかの組み合わせに対して実行され、2 つのビットストリングに対して実行される場合は、両方のビットストリングが同じ長さの必要があります。結果は、両方のオペランドがビット数の制約のない整数の場合は別のビット数の制約のない整数、それ以外の場合はビットストリング オペランドと同じ長さのビットストリングになります。これらの演算の正確な定義については、P.AppxD-14「*加算と減算*」を参照して下さい。

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

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

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

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

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

以下の動作をすることです。

- carry_in == '1' の場合、result == x-y となり、減算中に符号付きオーバフローが発生した場合は overflow == '1' となります。減算中に符号なしボローが発生しなかった場合（x >= y の場合）は carry_out == '1' となります。
- carry_in == '0' の場合、result == x-y-1 となり、減算中に符号付きオーバフローが発生した場合は overflow == '1' となります。減算中に符号なしボローが発生しなかった場合（x > y の場合）は carry_out == '1' となります。

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

飽和の擬似コードの詳細

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

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

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

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

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

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

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

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

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

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


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

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

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


A2.3 レジスタおよび実行状態

アプリケーションレベルのプログラマモデルには、アプリケーションプログラマが参照可能な汎用レジスタおよび専用レジスタの詳細が含まれています。データは ARM メモリモデルの制限に従ってメモリからレジスタにロードされるか、またはレジスタからメモリにストアされます（第 A3 章 *「ARM アーキテクチャのメモリモデル」*参照）。レジスタ内のデータはデータ処理命令によって変更されます。

A2.3.1 ARM コアレジスタ

13 個の汎用 32 ビットレジスタ (R0 ～ R12) 、および特別な名前と使用モデルを持つ追加の 3 つの 32 ビットレジスタが存在します。

SP スタックポインタ (R13) で、アクティブなスタックへのポインタとして使用されます。使用の制限については、P.A5-4「*レジスタ指定子として 0b1101 を使用する場合*」を参照して下さい。このレジスタは、リセット時にメインスタックの先頭にプリセットされます。詳細については、P.B1-7「*SP レジスタ*」を参照して下さい。

LR リンクレジスタ (R14) で、リンク付き分岐命令で開始されるサブルーチンからの復帰アドレスに関連する値（復帰リンク）がストアされるこのレジスタは、リセット時に不正な値（すべてのビットが 1）に設定されます。このリセット時の値を使用してサブルーチンからの復帰が試みられた場合、フォルト条件が発生します。

——— 注 ———

サブルーチンからの復帰をサポートするために R14 を必要としない場合、このレジスタを他の目的に使用できます。

PC プログラムカウンタ (R15) 。PC の使用モデルの詳細については、P.A5-3「*レジスタ指定子として 0b1111 を使用する場合*」を参照して下さい。PC には、リセット時にリセットハンドラ開始アドレスがロードされます。

ARM コアレジスタ演算の擬似コードの詳細

擬似コードでは、R[] 関数は次の目的に使用されます。

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

この関数には、次のようなプロトタイプがあります。

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

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

R[] 関数の詳細については、P.B1-10「*Thumb 命令セットで ARM コアレジスタへのアクセスを行う擬似コードの詳細*」を参照して下さい。PC にアドレスを書き込むと、そのアドレスへの単純な分岐、またはインターワーキング分岐が発生します。ARMv7-M では、インターワーキング分岐後に実行する命令セットとして、必ず Thumb 命令セットを選択する必要があります。

注

次の擬似コードは、ARMv7-M での動作を定義したものです。これは、以前の ARM アーキテクチャバリアントおよび他のプロファイルに適用される同等の擬似関数の定義よりも単純になっています。

単純な分岐は、BranchWritePC() 関数によって実行されます。

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

BranchWritePC(bits(32) address)
    BranchTo(address<31:1>:'0') ;
```

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

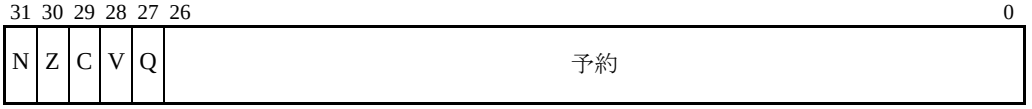
```
// BXWritePC()
// ===========

BXWritePC(bits(32) address)
    if address<0> == '1' then
        SelectInstrSet(InstrSet_Thumb) ;
        BranchTo(address<31:1>:'0') ;
    else
          ExceptionTaken(UsageFault) ;   // causes a UsageFault exception, recorded as 'Invalid
State'
```

LoadWritePC() および ALUWritePC() 関数は、二つの場合で使われ、それらは動作がアーキテクチャバージョン間でシステマチックに変更されるような場合です。M- プロファイルのアーキテクチャバリアントでは、これらの関数は分岐関数のエイリアスに単純化されます。

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

LoadWritePC(bits(32) address)
    BXWritePC(address) ;


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

ALUWritePC(bits(32) address)
    BranchWritePC(address) ;
```

A2.3.2 アプリケーションプログラム ステータスレジスタ (APSR)

プログラムステータスは、32 ビットのアプリケーションプログラム ステータスレジスタ (APSR) 内に記録されます。このレジスタの定義済みビットは、次のように一連のフラグに分けられます。



APSR のビットフィールドは、次の 2 つのカテゴリに分けられます。

- 予約ビットはシステム機能に割り当てられるか、または将来の拡張の際に使用可能です。現在予約に割り当てられているビットの詳細については、P.B1-7「*専用プログラム ステータスレジスタ (xPSR)*」を参照して下さい。アプリケーションレベルのソフトウェアでは、予約ビットから読み出された値を無視し、書き込み時にはレジスタの元の値を保存する必要があります。これらのビットは UNK/SBZP として定義されます。
- 次の各ビットは、多くの命令によりフラグとして設定されます。

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

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

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

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

  Q（ビット [27]） 結果の値の符号付または符号なしの範囲に対して、SSAT または USAT 命令が入力値を変更（飽和）すると 1 にセットされます。

A2.3.3 実行状態のサポート

ARMv7-M では Thumb 命令のみが実行されるため、常時 Thumb 状態で命令が実行されます。サポートされる命令のリストについては、第 A6 章 *「Thumb 命令の詳細」*を参照して下さい。

標準のプログラム実行に加えて、デバッグ状態が存在します。詳細については第 C1 章 *「ARMv7-M デバッグ」*を参照して下さい。

A2.3.4 特権実行

システムを適切に設計するには、アプリケーション開発者が基礎となるシステムアーキテクチャおよび提供されるサービスについて一定の知識を持っていることが必要です。システムサポートでは、一般的に特権操作と呼ばれるレベルのアクセスが必要になります。システムサポート コードにより、アプリケーションが特権と非特権のどちらで実行されるかが決定されます。オペレー

ティングシステムでは特権および非特権の両方がサポートされますが、アプリケーションは通常は非特権で実行されます。これによって、オペレーティングシステムは、システムリソースをアプリケーション専用、または共用として割り当て、他のプロセスおよびタスクに関して一定の保護を提供することができます。

スレッドモードは、ARMv7-M でのアプリケーション実行における基本モードです。スレッドモードはリセット時に選択され、システム環境に応じて特権または非特権で実行されます。特権実行では、多くの場合システムリソースの管理が必要になります。コードが非特権で実行されている場合は、スレッドモードから SVC 命令を実行してスーパバイザコール例外を生成することができます。スレッドモードの特権実行では、SVC 命令を使用してスーパバイザコールを生成するか、またはシステムアクセスおよび制御を直接処理することができます。

ハンドラモードでは、すべての例外は特権コードとして実行されます。ペリフェラルとの相互動作、メモリ割り当て、ソフトウェアスタックの管理などのリソース管理は、アプリケーションの代わりにスーパバイザコール ハンドラによって行われます。

特権実行、スレッドモードとハンドラモード、および例外の詳細については、第 B1 章 *「システム レベル プログラマモデル」*を参照して下さい。

A2.4 例外、フォルト、割り込み

例外は、例外生成命令の実行によって発生するか、または割り込み、メモリ管理保護違反、アライメントまたはバスフォルト、デバッグイベントなどのシステム動作に対する応答としてトリガされます。アーキテクチャ内では、同期例外と非同期例外が発生する可能性があります。

イベントの処理方法はシステムレベルの話題です。詳細については、P.B1-11「*例外モデル*」を参照して下さい。

A2.4.1 システム関連イベント

次のタイプの例外は、システムに関連したものです。例外が特定の命令に直接関係している場合は、関係する命令への参照が示されています。

スーパバイザコールはシステムの土台となるオペレーティングシステムへサービスを要求するために、アプリケーションコードが使用します。アプリケーションが SVC 命令を使用すると、システムへの特権アクセスを必要とするサービスを要求するスーパバイザコールが生成されます。次のようないくつかの形式のフォルトが発生する可能性があります。

- 命令の実行に関連するエラー
- ロード / ストアのいずれかにおけるデータメモリ アクセスエラー
- 実行状態に関連する各種のエラーに起因する UsageFault。未定義命令の実行は、UsageFault 例外の原因の一例です。
- デバッグイベントによる DebugMonitor 例外

一般にフォルトは、対応する実行中の命令に同期しています。一部のシステムエラーでは不正確な例外が発生し、この場合は例外が報告されるタイミングと、その例外を引き起こした命令との決まった関係は定義されていません。

割り込みは常に、プログラムフローに対する非同期イベントとして扱われます。システムタイマ (SysTick) 、非同期サービス呼び出し (PendSV[1]) 、および外部割り込み用のコントローラ (NVIC) はすべて定義されています。SysTick 割り込みの詳細については P.B2-8「*システムタイマ -SysTick*」を、割り込みコントローラの詳細については P.B2-9「*ネスト型ベクタ割り込みコントローラ (NVIC)*」を参照して下さい。

BKPT 命令は、デバッグイベントを生成します。詳細については P.C1-12「*デバッグイベントの動作*」を参照して下さい。

電力またはパフォーマンス上の理由から、アクションが完了したことをシステムへ通知するか、またはシステムに対してヒントを提供し、システムが現在のタスクの動作を一時停止できるようにすることが望ましい場合があります。この目的のため、次のような命令が用意されています。

- イベント送信およびイベント待機命令。詳細については、P.A6-211「*SEV*」および P.A6-275「*WFE*」を参照して下さい。
- 割り込み待ち。詳細については、P.A6-276「*WFI*」を参照して下さい。

1. サービス（システム）呼び出しは、土台となるオペレーティングシステムからのサービスを必要とするアプリケーションが使用します。PendSV に対応したサービス呼び出しは、その割り込みの処理時に実行されます。プログラム実行に同期して実行されるサービス呼び出しを行うには、SVC 命令を使用します。

A2.5 コプロセッサのサポート

ARMv7-M 実装では、オプションとしてコプロセッサをサポートできます。実装でサポートされていない場合は、すべてのコプロセッサが存在しないものとして扱われます。コプロセッサ 8 ～ 15 (CP8 ～ CP15) は、ARM によって予約されています。コプロセッサ 0 ～ 7 (CP0 ～ CP7) は実装定義で、命令セットアーキテクチャのコプロセッサ命令の制約に従います。

存在しない、または無効なコプロセッサに対してコプロセッサ命令が発行された場合、NOCP UsageFault が生成されます（P.B1-18「*フォルトの動作*」参照）。

有効なコプロセッサに対して不明な命令が発行された場合は、UNDEFINSTR UsageFault が生成されます。

第 A3 章
ARM アーキテクチャのメモリモデル

本章では、ARM メモリモデルに適用される一般的な原則について説明します。本章は以下のセクションから構成されています。

ARMv7-M はメモリマップドアーキテクチャです。ARMv7-M に適用されるアドレスマップ固有の詳細については、P.B2-2「*システム アドレスマップ*」で説明します。

A3.1 アドレス空間

ARM アーキテクチャでは、2^{32} バイトの単一でフラットなアドレス空間が使用されています。バイトアドレスは、0 ～ 2^{32} - 1 の範囲の符号なし整数として扱われます。

このアドレス空間は 2^{30} 個の 32 ビットワードで構成されているとみなされ、各アドレスはワードアラインド、つまりアドレスが 4 で割り切れます。ワードアラインドのアドレス A に存在するワードは、アドレス A、A + 1、A + 2、A + 3 にある 4 つのバイトで構成されます。また、アドレス空間は 2^{31} 個のハーフワード（16 ビット）とみなすことも可能で、その場合各アドレスはハーフワードアラインド、つまりアドレスが 2 で割り切れます。ハーフワードアラインドのアドレス A に存在するハーフワードは、アドレス A、A + 1 にある 2 つのバイトで構成されます。

命令フェッチは常にハーフワードアラインドですが、一部のロード / ストア命令ではアンアラインドアドレスがサポートされています。アンアラインドアドレスとは、アクセスのアドレス A について、ワードアクセスの場合には A[1:0]、ハーフワードアクセスの場合は A[0] が 0 以外の値になることを意味します。

アドレス計算は通常、普通の整数命令を使用して実行されます。これは、アドレス空間をオーバフローまたはアンダーフローした場合にラップアラウンドが行われることを意味します。これを別の観点から説明すると、すべてのアドレス計算はモジュロ 2^{32} に限定されるということです。

通常のシーケンシャルな命令実行の場合は、各命令の実行後に次の計算が行われ、次に実行される命令が決定されます。

```
(address_of_current_instruction) + (2 or 4) /* 16- and 32-bit instruction mix */
```

この計算がアドレス空間の最上位をオーバフローする場合、結果は予測不能です。ARMv7-M では、メモリの最上位には常に実行不可 (XN) メモリ属性が定義されているため、この条件は発生しません。詳細については、P.B2-2「*システム アドレスマップ*」を参照して下さい。このシナリオが発生すると、アクセス違反が報告されます。

上記の事項は、実行される命令にのみ適用されますが、これには条件コードチェックが不成立だった命令も含まれます。ほとんどの ARM 実装では、現在実行中の命令よりも先の命令のプリフェッチを行います。

LDC、LDM、LDRD、POP、PUSH、STC、STRD、STM の各命令は、メモリアドレスを加算して一連のワードにアクセスし、各レジスタのロード / ストアごとにメモリアドレスを 4 ずつインクリメントします。この計算がアドレス空間の最上位をオーバフローする場合、結果は予測不能です。

アンアラインドのロードまたはストアで、アドレス計算から 0xFFFFFFFF のバイトと 0x00000000 のバイトをアクセスすることになるメモリアクセスは、結果が予測不能です。

A3.1.1 仮想アドレシングと物理アドレシング

ARMv7-M では仮想メモリはサポートされていません。仮想アドレス (VA) は、常に物理アドレス (PA) に等しくなります。

A3.2 アライメントのサポート

システムアーキテクチャでは、ARMv7-M でのアライメントチェックに対して次の 2 つのポリシーのいずれかを選択できます。

- アンアラインドアクセスをサポートする。
- アンアラインドアクセスが発生した場合にフォルトを生成する。

ポリシーはアクセスのタイプに応じて変化します。ある実装では、すべてのアンアラインドアクセスに対してアライメントフォルトを強制的に生成するように構成することも可能です(下記参照)。

PC への書き込みは、P.A5-3「*レジスタ指定子として 0b1111 を使用する場合*」に示されている規則によって制限されます。

A3.2.1 アライメント動作

アドレスアライメントはデータアクセスと PC の更新に影響します。

アライメントおよびデータアクセス

次のデータアクセスでは、常時アライメントフォルトが生成されます。

- 非ハーフワードアラインドの LDREXH および STREXH
- 非ワードアラインドの LDREX および STREX
- 非ワードアラインドの LDRD、LDM{IA}、LDMDB、POP、LDC
- 非ワードアラインドの STRD、STM{IA}、STMDB、PUSH、STC

次のデータアクセスではアンアラインドなアドレシングがサポートされており、CCR.UNALIGN_TRP ビットがセットされている場合にのみアライメントフォルトが生成されます（P.B2-7「*システム制御ブロック (SCB)*」参照)。

- 非ハーフワードアラインドの LDRH、LDRHT、LDRSH、LDRSHT、STRH、STRHT
- 非ハーフワードアラインドの TBH
- 非ワードアラインドの LDR{T} および STR{T}

注

LDREXD および STREXD は ARMv7-M ではサポートされていません。
ストロングリオーダおよびデバイスのメモリタイプへのアクセスは、必ずそのデータサイズでアラインされている必要があります（P.A3-27「*メモリアクセスの制約事項*」参照)。

次の擬似コードは、ARMv7-M のアライメント動作の説明です。

AlignedAccessInstr() 、UnalignedAccess() 、AlignedAccess() 関数はローカルで、ここでの説明のみを目的として使用されています。実際のメモリアクセス機能については、命令定義（P.A6-17「*ARMv7-M Thumb 命令のアルファベット順リスト*」参照）で使用され、P.AppxD-28「*Mem*[]*」で定義されている MemU[]、MemA[]、MemAA[] を参照して下さい。

```
if address != Align(address, size) then  // the data access is to an unaligned address
    if AlignedAccessInstr() then         // the instruction does not support unaligned accesses
        UFSR.UNALIGNED = '1';            // update the UsageFault Status Register
        ExceptionTaken(UsageFault) ;      // UsageFault exception taken
    else
        if CCR.UNALIGN_TRP then          // Configuration and Control Register - trap on all
                                         // ... unaligned accesses
            UFSR.UNALIGNED = '1';        // update the UsageFault Status Register
            ExceptionTaken(UsageFault) ;  // UsageFault exception taken
        else
            UnalignedAccess(Address) ;    // perform an unaligned access
else
    AlignedAccess(Address) ;              // perform an aligned access
```

アライメントと PC の更新

すべての命令フェッチは、ハーフワードアラインドの必要があります。すべての例外復帰の違反は例外復帰機構により INVSTATE または INVPC 用法フォルトとして捕捉されます。

例外エントリおよび復帰の場合は、次の処理が行われます。

- ビット [0] がゼロのベクタを使用して例外に入った場合、INVSTATE 用法フォルトが発生します。
- 予約されている EXC_RETURN 値は、INVPC 用法フォルトを発生します。
- 例外復帰時にスタックから PC へアラインされていない値がロードされた場合、結果は予測不能です。

PC が更新される他のすべての場合は、次の処理が行われます。

- ADD または MOV 命令で PC[1] へのロードを行う場合、値のビット [0] は無視されます。
- PC への BLX、BX、LDR 、あるいは PC を含む POP または LDM では、PC に書き込まれた値のビット [0] が 0 の場合に INVSTATE 用法フォルトが発生します。
- アドレスがワードアラインドでないメモリ位置から PC へ値をロードした場合、結果は予測不能です。

1. 16 ビット形式の ADD（レジスタ）および MOV（レジスタ）命令の場合のみ。それ以外の場合、PC へのロードの結果は予測不能です。

A3.3 エンディアンのサポート

アドレス空間の規則（P.A3-2「*アドレス空間*」）により、ワードアラインドのアドレス A は次のように定義されます。

- アドレス A のワードは、アドレス A、A + 1、A + 2、A + 3 のバイトから構成されます。
- アドレス A のハーフワードは、アドレス A および A + 1 のバイトから構成されます。
- アドレス A + 2 のハーフワードは、アドレス A + 2 および A + 3 のバイトから構成されます。
- 結果として、アドレス A のワードは、アドレス A および A + 2 のハーフワードから構成されます。

ただし、この定義ではワード、ハーフワード、バイト間のマッピングが完全には指定されていません。メモリシステムでは、次のマッピング方式のいずれかが使用されます。この選択は、メモリシステムのエンディアン形式と呼ばれます。

リトルエンディアンのメモリシステムでは、メモリ上のバイトは ARM レジスタで表 A3-1 に示すように解釈します。

- アドレス A のバイトまたはハーフワードは、そのアドレスにあるワード内の最下位バイトまたはハーフワードです。
- アドレス A のバイトは、そのアドレスにあるハーフワード内の最下位バイトです。

表 A3-1 リトルエンディアンのバイトフォーマット

	31 24	23 16	15 8	7 0
アドレス A のワード	バイト {Addr + 3}	バイト {Addr + 2}	バイト {Addr + 1}	バイト {Addr + 0}
アドレス A のハーフワード			バイト {Addr + 1}	バイト {Addr + 0}

ビッグエンディアンのメモリシステムでは、メモリ上のバイトは ARM レジスタで表 A3-2 に示すように解釈されます。

- アドレス A のバイトまたはハーフワードは、そのアドレスにあるワード内の最上位バイトまたはハーフワードです。
- アドレス A のバイトは、そのアドレスにあるハーフワード内の最上位バイトです。

表 A3-2 ビッグエンディアンのバイトフォーマット

	31 24	23 16	15 8	7 0
アドレス A のワード	バイト {Addr + 0}	バイト {Addr + 1}	バイト {Addr + 2}	バイト {Addr + 3}
アドレス A のハーフワード			バイト {Addr + 0}	バイト {Addr + 1}

ワードアドレス A に対して、アドレス A のワード、アドレス A および A + 2 のハーフワード、およびアドレス A、A + 1、A + 2、A + 3 のバイトが各エンディアン形式においてどのようにマップされるかを表 A3-3 および表 A3-4 に示します。

表 A3-3 リトルエンディアンのメモリシステム

MSByte	MSByte -1	LSByte + 1	LSByte
アドレス A のワード			
アドレス A + 2 のハーフワード		アドレス A のハーフワード	
アドレス A + 3 のバイト	アドレス A + 2 のバイト	アドレス A + 1 のバイト	アドレス A のバイト

表 A3-4 ビッグエンディアンのメモリシステム

MSByte	MSByte -1	LSByte + 1	LSByte
アドレス A のワード			
アドレス A のハーフワード		アドレス A + 2 のハーフワード	
アドレス A のバイト	アドレス A + 1 のバイト	アドレス A + 2 のバイト	アドレス A + 3 のバイト

ビッグエンディアンおよびリトルエンディアンのマッピング方式により、ワードまたはハーフワードのバイトが解釈される順序が決定されます。

例えば、アドレス 0x1000 からのワード（4 バイト）のロードでは、使用されているマッピング方式に関わりなく、メモリ位置 0x1000、0x1001、0x1002、0x1003 に含まれているバイトがアクセスされます。マッピング方式により、それらのバイトの上位下位が決定されます。

A3.3.1 ARMv7-M でのエンディアンマッピングの制御

ARMv7-M ではエンディアンモデルを選択可能で、システムリセット時の制御入力によってビッグエンディアン (BE) またはリトルエンディアン (LE) として構成されます。エンディアンマッピングには、次の制約事項があります。

- エンディアン設定はデータアクセスのみに適用され、命令フェッチは常にリトルエンディアンです。
- システム制御空間（P.B2-6「*システム制御空間 (SCS)*」）へのロード / ストアは、常にリトルエンディアンです。

ARMv7-M でビッグエンディアン命令フォーマットをサポートする必要がある場合、バス構造でハーフワード内のバイトをスワップする必要があります。このバイトスワップは命令フェッチのみに必要となるもので、データアクセス時には発生しないことが要求されます。

例えば、32 ビットバスでの命令フェッチの場合は次の処理を行います。

```
PrefetchInstr<31:24> -> PrefetchInstr<23:16>
PrefetchInstr<23:16> -> PrefetchInstr<31:24>
PrefetchInstr<15:8> -> PrefetchInstr<7:0>
PrefetchInstr<7:0> -> PrefetchInstr<15:8>
```


命令アライメントおよびバイトの順序付け

Thumb-2 では、すべての命令で 16 ビットアライメントが使用されます。つまり、32 ビットの命令は 2 つのハーフワード hw1 および hw2 として扱われ、hw1 が下位アドレスになります。

命令のエンコードダイアグラムでは、hw1 は hw2 の左側に表示されます。これにより、エンコードダイアグラムがより自然に表示されます。32 ビット Thumb 命令のバイト順序を図 A3-1 に示します。

図 A3-1 メモリ内での命令のバイト順序

A3.3.2 エレメントサイズとエンディアン形式

データのエンディアン形式マッピングは、ロード / ストア命令で転送されるデータ要素のサイズに適用されます。各ロード / ストア命令のデータ要素サイズを表 A3-5 に示します。

表 A3-5 ロード / ストア命令とエレメントサイズの関係

命令クラス	命令	データ要素のサイズ
バイトのロード / ストア	LDR{S}B{T}、STRB{T}、LDREXB、STREXB	バイト
ハーフワードのロード / ストア	LDR{S}H{T}、STRH{T}、TBH、LDREXH、STREXH	ハーフワード
ワードのロード / ストア	LDR{T}、STR{T}、LDREX、STREX	ワード
2 ワードのロード / ストア	LDRD、STRD	ワード
複数ワードのロード / ストア	LDM{IA,DB}、STM{IA,DB}、PUSH、POP、LDC、STC	ワード

A3.3.3 汎用レジスタ内でバイトを反転する命令

アプリケーションまたはデバイスドライバが、内部データ構造やオペレーティングシステムとはエンディアン形式が異なる、メモリマップされた周辺装置のレジスタや共有メモリ構造とのインタフェースとして動作する必要がある場合、データのエンディアン形式を明示的に変換するための効率的な手段が必要です。

Thumb-2 には、次に示すバイト変換用の命令が用意されています（詳細については、第 A6 章 *「Thumb 命令の詳細」* にある命令の定義を参照して下さい）。

REV ワード（4 バイト）レジスタでの 32 ビットの表現を反転します。

REVSH ハーフワードでの符号付き 16 ビットの表現を反転し、符号拡張を行います。

REV16 レジスタ内のパックドハーフワードについて、符号なし 16 ビットの表現を反転します。

A3.4 同期化とセマフォ

排他アクセス命令はノンブロッキングの共有メモリ同期化基本命令をサポートし、セマフォの読み出しと書き込みの間で計算を実行可能で、マルチプロセッサシステム設計用にスケーリング可能です。

ARMv7-M で用意されている同期化基本命令は、次のとおりです。

- 排他ロード
  - LDREX P.A6-107「*LDREX*」を参照して下さい。
  - LDREXB P.A6-108「*LDREXB*」を参照して下さい。
  - LDREXH P.A6-109「*LDREXH*」を参照して下さい。
- 排他ストア
  - STREX P.A6-233「*STREX*」を参照して下さい。
  - STREXB P.A6-234「*STREXB*」を参照して下さい。
  - STREXH P.A6-235「*STREXH*」を参照して下さい。
- 排他クリア CLREXP.A6-56「*CLREX*」を参照して下さい。

注

このセクションでは、LDREX および STREX 命令など、同期化基本命令のペアである排他ロード / 排他ストア操作について説明します。同じ説明は、他の同期化基本命令のペアについても適用されます。

- LDREXB STREXB とともに使用します。
- LDREXH STREXH とともに使用します。

各排他ロード命令は、対応する排他ストア命令とペアでのみ使用できます。

STREXD および LDREXD は ARMv7-M ではサポートされていません。

排他ロード / 排他ストア命令ペアの使用法のモデルでは、メモリアドレス x への読み出し / 書き込みは次のように行われます。

- 排他ロード命令は、メモリアドレス x からの値を常に正常に読み出します。
- 対応する排他ストア命令は、排他ロードによる読み出しの後に、他のプロセッサまたはプロセスがアドレス x へストアを実行していない場合に限り、メモリアドレス x への書き込みに成功します。排他ストア操作では、メモリ書き込みに成功したかどうかを示すステータスビットが返されます。

排他ロード命令では、メモリの小さいブロックに対して、排他アクセスを示すタグが付けられます。タグ付けされるブロックのサイズは実装定義です。詳細については P.A3-16「*タグ付けとタグ付けされたメモリブロックのサイズ*」を参照して下さい。このタグは、同じアドレスへの排他ストア命令によりクリアされます。

A3.4.1 排他アクセス命令と共有不可メモリ領域

*共有可*属性を持たないメモリ領域の場合、排他アクセス命令は、プロセッサが排他ロードを実行するアドレスにタグを付ける *ローカルモニタ*に依存します。同じプロセッサがいずれかのアドレスを変更するために排他ストアを使うアクセスをすると、アボートされない限り、そのタグがクリアされることが保証されています。

排他ロードは、メモリからロードを行うとともに次の処理を実行します。

- 実行中のプロセッサにより、その物理メモリアドレスを排他アクセスとしてタグ付けが行われます。
- 実行中のプロセッサのローカルモニタが、排他アクセス状態へ遷移します。

排他ストア命令によりメモリに対して条件付きストアが実行されます。このとき、ローカルモニタの状態に応じて次の動作が行われます。

ローカルモニタが排他アクセス状態の場合

- 排他ストアのアドレスが、以前の排他ロードによってモニタ内にタグ付けされたアドレスと同じである場合はストアが実行されます。それ以外の場合、ストアが実行されるかどうかは実装定義です。
- 次のように、ステータス値がレジスタに返されます。
  - ストアが実行された場合、0
  - それ以外の場合、1
- 実行中のプロセッサのローカルモニタが、オープンアクセス状態へ遷移します。

ローカルモニタがオープンアクセス状態の場合

- ストアは実行されません。
- ステータス値 1 がレジスタに返されます。
- ローカルモニタはオープンアクセス状態のままです。

ステータス値が返されるレジスタは、排他ストア命令によって定義されます。
プロセッサが排他ストア以外の命令を使用して書き込みを行う場合、次の動作が行われます。

- ローカルモニタが処理を担当しない物理アドレスに書き込みが行われる場合、その書き込みによってローカルモニタの状態は影響を受けません。
- ローカルモニタが処理を担当する物理アドレスに書き込みが行われる場合、その書き込みによってローカルモニタの状態が影響を受けるかどうかは実装定義です。

ローカルモニタが排他アクセス状態で、プロセッサが最後に実行した排他ロードのアドレス以外の共有不可メモリのアドレスへ排他ストアが実行される場合は、ストアが成功するかどうかは実装定義ですが、ローカルモニタは必ずオープンアクセス状態にリセットされます。ARMv7-M では、この機構はすべての場合においてソフトウェアプログラミング エラーとして扱う必要があります。

注

共有不可メモリでは、タグ付けされた物理アドレスへのストアがタグのクリアを引き起こすかどうかは、そのストアが物理アドレスのタグ付けを引き起こしたオブザーバ以外のオブザーバによるものだとすると、実装依存になります。

ローカルモニタのステートマシンを P.A3-11 図 A3-2 に示します。図に示されている各操作の影響を P.A3-11 表 A3-6 に示します。

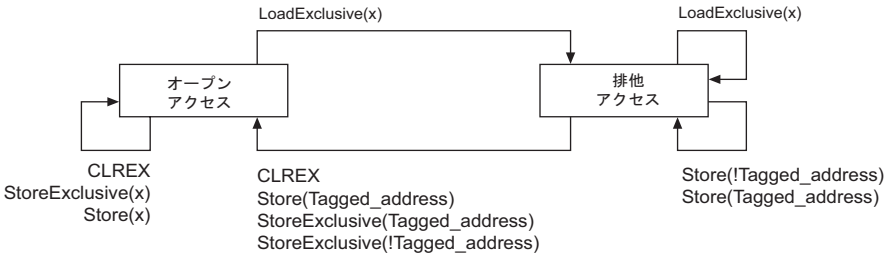


図 A3-2 ローカルモニタのステートマシン ダイアグラム

注

- ローカルモニタの実装定義オプションは構築されているローカルモニタと整合性があるので、物理アドレスを保持せず、代わりにどのアクセスも以前の LDREX のアドレスと一致するものとして扱います。
- ローカルモニタの実装では、他のプロセッサからの排他ロード / 排他ストア操作を認識する必要はありません。
- STR または STREX が別のオブザーバから実行される場合、排他アクセスからオープンアクセス状態への遷移が発生するかどうかは予測不能です。

表 A3-6 に示されている動作の影響を図 A3-2 に示します。

表 A3-6 ローカルモニタ上での排他命令および書き込み動作の影響

初期状態	操作[a]	影響	最終状態
オープンアクセス	CLREX	影響なし。	オープンアクセス
オープンアクセス	StoreExcl(x)	メモリを更新せずに、ステータス 1 を返す。	オープンアクセス
オープンアクセス	LoadExcl(x)	メモリから値をロードして、アドレス x にタグを付ける。	排他アクセス
オープンアクセス	Store(x)	メモリを更新するが、モニタには影響なし。	オープンアクセス
排他アクセス	CLREX	タグを付けられたアドレスをクリアする。	オープンアクセス
排他アクセス	StoreExcl(t)	メモリを更新して、ステータス 0 を返す。	オープンアクセス

表 A3-6 ローカルモニタ上での排他命令および書き込み動作の影響

初期状態	操作[a]	影響	最終状態
排他アクセス	StoreExcl(!t)	メモリを更新して、ステータス 0 を返す。[b]	オープンアクセス
		メモリを更新せずに、ステータス 1 を返す。[b]	
排他アクセス	LoadExcl(x)	メモリから値をロードして、アドレスへのタグを x に変更する。	排他アクセス
排他アクセス	Store(!t)	メモリを更新するが、モニタには影響なし。	排他アクセス
排他アクセス	Store(t)	メモリを更新する。	排他アクセス[b]
			オープンアクセス[b]

a. この表の表記は次の意味です。
LoadExcl は、任意の排他ロード命令を表します。
StoreExcl は、任意の排他ストア命令を表します。
Store は、排他ストア操作以外の任意のストア操作を表します。
t はタグ付けされたアドレスで、最後の排他ロード命令のアドレスのビット [31:a] を表します。詳細については、P.A3-16「*タグ付けとタグ付けされたメモリブロックのサイズ*」を参照して下さい。

b. 実装定義の代替動作。

A3.4.2 排他アクセス命令と共有可メモリ領域

*共有可*属性を持つメモリ領域の場合、排他アクセス命令は次の項目に依存します。

- システム内の各プロセッサに対する *ローカルモニタ*。これにより、プロセッサが排他ロードを実行するアドレスにタグが付けられます。ローカルモニタは P.A3-9「*排他アクセス命令と共有不可メモリ領域*」で説明されているように動作し、システム内の他のプロセッサからの排他アクセスを無視できます。
- 単一の *グローバルモニタ*。これにより、ある特定のプロセッサのために物理アドレスに排他アクセスのタグが付けられます。このタグは以後、そのアドレスに対する排他ストアがおきたかどうかの判定に使用されます。タグ付けされたアドレスに対して、他のオブザーバによる変更が試みられ、その操作がアボートしなかった場合は、必ずタグがクリアされます。システム内の各プロセッサについて、グローバルモニタは次の処理を実行します。
  — タグ付けされた単一アドレスの保持
  — 排他アクセス ステートマシンの保守

グローバルモニタは、あるプロセッサブロック内にあるか、またはメモリインタフェースの 2 次モニタとして存在します。

実装では、グローバルモニタとローカルモニタの機能を単一のユニットに結合することができます。

グローバルモニタの動作

*共有可*メモリからの排他ロードを実行した場合、メモリからのロードが行われるとともに、アクセスされた物理アドレスには、要求を発行したプロセッサに対する排他アクセスのタグが付けられます。同時に、このアクセスは、その要求をしているプロセッサによってタグ付けされていた他の物理アドレスでの排他アクセスタグを削除します。グローバルモニタでは、プロセッサごとに共有可メモリへのひとつの排他アクセストランザクションのみがサポートされています。

排他ストアは、メモリへの条件付きストアを次のように実行します。

- アクセスされる物理アドレスに、要求を発行したプロセッサに対する排他アクセスのタグが付けられており、ローカルモニタおよびグローバルモニタのステートマシンがいずれも排他アクセス状態の場合にのみ、ストアの成功が保証されます。この場合、次の動作が行われます。
  - ステータス値 0 がレジスタに返され、ストアの成功を知らせます。
  - 要求を発行したプロセッサに対するグローバルモニタのステートマシンの最終状態は、実装定義です。
  - アクセスされるアドレスが、グローバルモニタのステートマシン内で、他のプロセッサに対する排他アクセスとしてタグ付けされている場合、そのステートマシンはオープンアクセス状態へ遷移します。
- 要求を発行したプロセッサに対して排他アクセスとしてタグ付けされているアドレスがない場合、ストアは成功せず、次の動作が行われます。
  - ステータス値 1 がレジスタに返され、ストアが失敗したことを示します。
  - グローバルモニタは影響を受けず、要求を発行したプロセッサに対してオープンアクセス状態が維持されます。
- 要求を発行したプロセッサに対して、排他アクセスとして別の物理アドレスにタグが付けられている場合、ストアが成功するかどうかは実装定義です。
  - ストアが成功した場合はステータス値 0 が、失敗した場合は値 1 がレジスタに返されます。
  - プロセッサに対するグローバルモニタのステートマシンが、排他ストアの前に排他アクセス状態だった場合、そのステートマシンがオープンアクセス状態に遷移するかどうかは実装定義です。

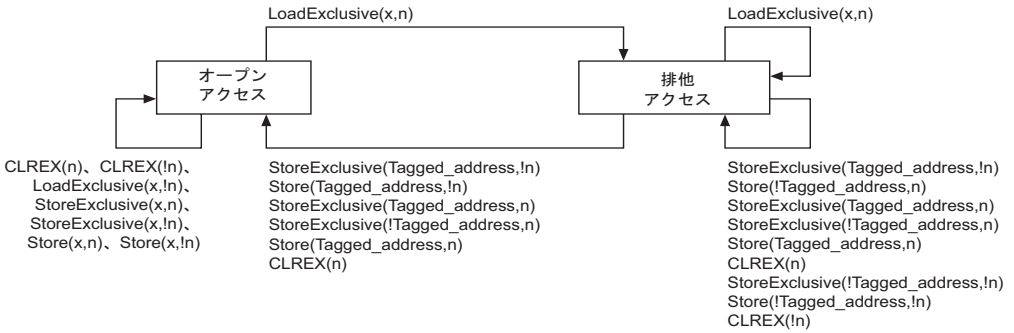
ステータス値が返されるレジスタは、排他ストア命令によって定義されます。

共有メモリシステムでは、グローバルモニタには、システム内の各プロセッサに対して個別のステートマシンが実装されます。プロセッサ (n) による共有可のメモリへのアクセスに対するステートマシンは、参照可能なすべての共有可メモリアクセスに応答することができます。これは、次のアクセスに応答することを意味します。

- 対応するプロセッサ (n) によって生成されるアクセス
- 共有メモリシステム (!n) で、他のオブザーバによって生成されるアクセス

共有メモリシステムでは、グローバルモニタは、システム中の排他ロードまたは排他ストアを生成する可能性のあるオブザーバごとに別々のステートマシンを実装します。

グローバルモニタのプロセッサ (n) に対するステートマシンを P.A3-14 図 A3-3 に示します。図に示されている各操作の影響を P.A3-14 表 A3-7 に示します。



‡ STREXがメモリを更新する場合にのみ、STREX(Tagged_Address,!n)はモニタをクリアします

斜体で示されている操作は、使用可能な代替実装定義オプションを示します

この図の用語はそれぞれ次の意味です　LoadExclusiveは、任意の排他ロード命令を表します
StoreExclusiveは、任意の排他ストア命令を表します
Storeは、他の任意のストア命令を表します

LoadExclusive操作により、タグ付けされたアドレスは、操作で使用されているアドレスxの最上位ビットに更新されます。
詳細については、「タグ付けされたメモリブロックのサイズ」セクションを参照して下さい

図 A3-3 マルチプロセッサシステムにおけるプロセッサ (n) に対するグローバルモニタのステートマシンダイアグラム

注

- 排他ストアによりメモリが正常に更新されるかどうかは、アクセスされたアドレスが、排他ストア命令を発行するプロセッサに対してタグ付けされている共有可のメモリアドレスと一致するかどうかに関係します。この理由のため、(!n) エントリがプロセッサ (n) のステートマシンの状態遷移を発生させる方法のみを図 A3-3 および表 A3-7 に示します。
- 排他ロードでは、その排他ロード命令を発行するプロセッサに対してタグ付けされている共有可のメモリアドレスのみを更新できます。
- グローバルモニタの CLREX 命令の影響は実装定義です。

表 A3-7 に示されている操作の影響図 A3-3 に示します。

表 A3-7 プロセッサ (n) に対するグローバルモニタのロード / ストア操作の影響

初期状態[a]	操作[b]	影響	最終状態[a]
オープン	CLREX(n), CLREX(!n)	なし	オープン
オープン	StoreExcl(x,n)	メモリを更新せずに、ステータス 1 を返す。	オープン

表 A3-7 プロセッサ (n) に対するグローバルモニタのロード / ストア操作の影響

初期状態[a]	操作[b]	影響	最終状態[a]
オープン	LoadExcl(x,!n)	メモリから値をロードするが、プロセッサ (n) に対するタグアドレスには影響なし。	オープン
オープン	StoreExcl(x,!n)	STREX を発行するプロセッサに対するステートマシンおよびタグアドレスに依存。[c]	オープン
オープン	STR(x,n), STR(x,!n)	メモリを更新するが、モニタには影響なし。	オープン
オープン	LoadExcl(x,n)	メモリから値をロードし、アドレス x にタグを付ける。	排他
排他	LoadExcl(x,n)	メモリから値をロードし、アドレス x にタグを付ける。	排他
排他	CLREX(n)	なし。最終状態は実装定義です。	排他[e]
			オープン[e]
排他	CLREX(!n)	なし	排他
排他	StoreExcl(t,!n)	メモリを更新して、ステータス 0 を返す。[c]	オープン
		メモリを更新せずに、ステータス 1 を返す。[c]	排他
排他	StoreExcl(t,n)	メモリを更新して、ステータス 0 を返す。[d]	オープン
			排他
排他	StoreExcl(!t,n)	メモリを更新して、ステータス 0 を返す。[e]	オープン
			排他
		メモリを更新せずに、ステータス 1 を返す。[e]	オープン
			排他
Exclusive	StoreExcl(!t,!n)	STREX を発行するプロセッサに対するステートマシンおよびタグアドレスに依存	排他
排他	Store(t,n)	メモリを更新する。	排他[e]
			オープン[e]
排他	Store(t,!n)	メモリを更新する。	オープン
排他	Store(!t,n), Store(!t,!n)	メモリを更新するが、モニタには影響なし。	排他

a. オープン = オープンアクセス状態、排他 = 排他アクセス状態。

b. この表の表記は次の意味です。
   LoadExcl は、任意の排他ロード命令を表します。
   StoreExcl は、任意の排他ストア命令を表します。
   Store は、排他ストア以外の任意のストア操作を表します。
   t はプロセッサ (n) に対してタグ付けされたアドレスで、プロセッサ (n) によって発行された最後の排他ロード命令のアドレスのビット [31:a] を表します。詳細については、*タグ付けとタグ付けされたメモリブロックのサイズ*を参照して下さい。
c. STREX(x,!n) または STREX(t,!n) 演算の結果は、STREX 命令を発行するプロセッサに対するステートマシン、およびタグ付けされたアドレスに関係します。この表では、可能な結果のそれぞれについて、プロセッサ (n) のステートマシンがどのような影響を受けるかを示しています。
d. タグ付けされたアドレスへの STREX が成功した場合、ステートマシンの状態は実装定義です。ただし、この状態はグローバルモニタの次の操作には影響を与えません。
e. 作用は実装定義です。この表では、許可されている実装をすべて示しています。

A3.4.3 タグ付けとタグ付けされたメモリブロックのサイズ

P.A3-11 図 A3-2 および P.A3-14 図 A3-3 に示されているように、LDREX 命令が実行されるとき、結果のタグアドレスではメモリアドレスの最下位ビットが無視されます。

```
Tagged_address == Memory_address<31:a>
```

この式の a この式の実装定義です。例えば、a = 4 という実装の場合、アドレス 0x000341B4 に対する LDREX 命令が成功すると、アドレスのビット [31:4] である 0x000341B がタグの値になります。0x000341B0 ～ 0x000341BF の 4 ワードのメモリに、排他アクセスのタグが付けられたことを意味します。これ以後に、このブロックに含まれるいずれかのアドレスに対して有効な STREX が実行されると、このタグは削除されます。

したがって、タグが付けられたメモリブロックのサイズは、次の範囲で実装定義です。

- 1 ワード。a = 2 の実装
- 512 ワード。a = 11 の実装

A3.4.4 コンテキストスイッチのサポート

ローカルモニタは、コンテキストスイッチ後にオープンアクセス状態にあることを保証する必要があります。ARMv7-M では、ローカルモニタは例外エントリまたは例外終了シーケンスの一部として、自動的にオープンアクセスに変更されます。ローカルモニタは、CLREX 命令によって強制的にオープンアクセス状態にすることもできます。

注

コンテキストスイッチは、アプリケーションレベルの操作ではありません。ただし、この情報は排他操作の説明の一部としてここで紹介されています。

コンテキストスイッチが発生すると、以後に実行される排他ストアが失敗する可能性があり、その場合にはロード … ストアのシーケンスを再実行する必要があります。この条件が発生する可能性を最小限にするため、排他ストア命令は関連する排他ロード命令のできるだけ近くに配置することをお勧めします。詳細については、P.A3-17「*排他ロード/排他ストアの使用に関する制約事項*」を参照して下さい。

A3.4.5 排他ロード / 排他ストアの使用に関する制約事項

排他ロード / 排他ストア命令は、1 つのペアとして動作するように設計されています。例として、LDREX/STREX のペアや LDREXB/STREXB のペアが挙げられます。P.A3-16「*コンテキストスイッチのサポート*」で説明されているように、排他ストア命令は常に関連する排他ロード命令から数命令以内に配置することをお勧めします。これらの関数の別の実装をサポートするため、ここで示されている注意事項および制約事項に従うようにソフトウェアを設計する必要があります。

これらの注意事項では LDREX/STREX のペアの使用法について説明されていますが、同様に他の排他ロード / 排他ストアのペアにも適用されます。

- 排他命令は、実行される各プロセッサスレッドに対して、ひとつの排他アクセストランザクションをサポートしています。アーキテクチャはこの排他アクセスを使用することで、IsExclusiveLocal() 関数の一部としてアドレスおよびサイズのチェックを行う必要がなくなります。同じ実行スレッド内で STREX のターゲットアドレスが前の LDREX のアドレスと異なる場合、動作は予測不能です。結果的に、LDREX/STREX のペアは同じアドレスで実行されている場合のみ最終的に成功することが保証されます。
- メモリへの明示的なストアにより他のプロセッサに関連する排他モニタのクリアが実行される可能性があるため、LDREX と STREX との間でのストアの実行はライブロック状況を引き起こす場合があります。このため、単一のコードシーケンスのコードでは LDREX と STREX との間に明示的なストアを置くことは避ける必要があります。
- 2 つの STREX 命令を、間に LDREX 命令を入れないで実行すると、2 番目の STREX によりステータス値 1 が返されます。これは次のことを意味します。
  - ある実行スレッド中では、すべての STREX 命令の前には、その命令に対応する LDREX 命令が配置される必要があります。
  - LDREX 命令の後に、必ず STREX 命令を配置する必要はありません。
- 排他ロード / 排他ストア命令の実装によっては、いかなる実行スレッドにおいても、排他ストアのトランザクションサイズが、そのスレッドで実行された前の排他ロードのトランザクションサイズと同じであることを要求する可能性があります。同じ実行スレッド内で排他ストアのトランザクションサイズが前の排他ロードのトランザクションサイズと異なる場合、動作は予測不能です。結果的に、ソフトウェアでは排他ロード / 排他ストアのペアが同じアドレスで実行されている場合のみ、操作の最終的な成功を前提にできます。
- 実装によっては、アプリケーション関連の原因がなくても LDREX と STREX との間の排他モニタがクリアされる場合があります。例えば、キャッシュの退出によってクリアが発生する場合があります。そのような実装用に記述されたコードでは、LDREX 命令と STREX 命令との間で明示的なメモリアクセスまたはキャッシュ保守操作を避ける必要があります。
- 実装によっては、単一のコードシーケンス内で LDREX 操作と STREX 操作をすぐ近くに配置することで利益が得られる場合があります。これにより、LDREX 命令と STREX 命令との間で排他モニタ状態がクリアされる可能性を最小限にすることができます。このため、最良の性能を得るために、単一のコードシーケンス内では LDREX 命令と STREX 命令の間が 128 バイト以内になるように制限を設けることを強くお勧めします。
- コヒーレントプロトコルを実装しているか、または単一のマスタのみが存在する実装では、あるプロセッサに対するローカルモニタとグローバルモニタとが結合されている場合があります。定義で実装定義および予測不能と記載されている部分は、このような動作に対応しています。

- アーキテクチャでは、排他としてマークされる領域サイズの上限は 2048 バイトに設定されます。そのため、性能上の理由から、ソフトウェアを作成するとき、排他アクセスによってアクセスされるオブジェクトは最大でも 2048 バイトの単位に分割することをお勧めします。これは、機能的な要件ではなく性能上のガイドラインです。
- LDREX 操作と STREX 操作は、ノーマルメモリ属性を持つメモリ上でのみ実行できます。
- LDREX/STREX のペアによってアクセスされるメモリのメモリ属性が LDREX と STREX との間で変更されると、動作は予測不能になります。

A3.4.6 同期化基本命令とメモリオーダ モデル

同期化基本命令は、その命令によってアクセスされるメモリタイプのメモリオーダ モデルに従います。この理由のため、次の処理が必要になります。

- スピンロックを要求する移植可能なコードでは、スピンロックの要求とスピンロックを使用するアクセス実行との間に DMB 命令が含まれている必要があります。
- スピンロックを解放する移植可能なコードでは、スピンロックをクリアする記述の前に DMB 命令が含まれている必要があります。

この要件は、排他ロード / 排他ストア命令のペア、例えば LDREX/STREX を使用するコードに適用されます。

A3.5 メモリのタイプおよび属性とメモリオーダ モデル

ARMv7-M では、システム メモリマップ内でメモリおよびデバイスをサポートするために必要な特性を持つ、メモリ属性のセットが定義されています。

メモリオーダ モデルと呼ばれるメモリ領域に対するアクセスの順序付けは、メモリ属性によって定義されます。このモデルについては、以下のセクションで説明します。

- *メモリタイプ*
- *ARMv7 のメモリ属性の要約* : P.A3-20
- *ARM アーキテクチャにおけるアトミック性* : P.A3-21
- *ノーマルメモリ* : P.A3-23
- *デバイスメモリ* : P.A3-25
- *ストロングリオーダメモリ* : P.A3-26
- *メモリアクセスの制約事項* : P.A3-27

A3.5.1 メモリタイプ

各メモリ領域に対して、最も重要なメモリの特性がメモリタイプを決めます。メモリタイプには次の 3 つが存在し、これらは相互に排他です。

- ノーマル
- デバイス
- ストロングリオーダ

ノーマルおよびデバイスのメモリ領域には、追加の属性が存在します。

一般に、プログラムコードおよびデータ記憶域に使用されるメモリはノーマルメモリです。ノーマルメモリテクノロジの例には、次のものがあります。

- プログラムされたフラッシュ ROM

  注

  プログラミングの間、フラッシュメモリはノーマルメモリよりさらに厳密に順番を守る必要があります。

- ROM
- SRAM
- DRAM および DDR メモリ

システムペリフェラル (I/O) は、一般にノーマルメモリとは別のアクセス規則に準拠します。I/O アクセスの例には、次のものがあります。

- 連続アクセスにより次の処理が行われる FIFO
  - 書き込みアクセスで、キューに値を追加する。
  - 読み出しアクセスで、キューから値を削除する。
- 割り込みコントローラレジスタ。アクセスが割り込み応答として使用され、コントローラ自身の状態を変化させます。

- ノーマルメモリ領域のタイミングおよび正確さの設定に使用されるメモリコントローラ構成レジスタ
- メモリマップされたペリフェラルで、メモリ位置へのアクセスにより、システム内に副作用が引き起こされる場合

ARMv7 では、これらのアクセス用のメモリマップ領域は、デバイスまたはストロングリオーダメモリとして定義されます。システムの正確さを保証するために、デバイスおよびストロングリオーダメモリのアクセス規則は、ノーマルメモリの規則よりも厳格になっています。

- 読み出し / 書き込みアクセスのいずれにも、副作用の可能性があります。
- 例外からの復帰時などに、アクセスを繰り返し行うことはできません。
- アクセスの数、順序、サイズを維持する必要があります。

さらに、ストロングリオーダメモリの場合は、すべてのメモリアクセスは厳密に順序付けられており、メモリアクセス命令のプログラム上の順序に対応しています。

A3.5.2 ARMv7 のメモリ属性の要約

メモリ属性の要約を表 A3-8 に示します。これらの属性の詳細については、以下を参照して下さい。

- P.A3-23「*ノーマルメモリ*」および P.A3-26「*デバイスメモリ領域に対する共有可属性*」（*共有可*属性の場合）
- P.A3-24「*ライトスルーキャッシュ可、ライトバックキャッシュ可、およびキャッシュ不可のノーマルメモリ*」（*キャッシュ可*属性の場合）

表 A3-8 メモリ属性の要約

メモリタイプ属性	共有可否	その他の属性	説明
ストロングリオーダ	-		ストロングリオーダメモリに対するメモリアクセスは、すべてプログラム順に発生します。ストロングリオーダ領域は、すべて共有可であることを前提としています。
デバイス	共有可		複数のプロセッサにより共有されるメモリマップされたペリフェラルを処理することを意図しています。
	共有不可		単一プロセッサによってのみ使用されるメモリマップされたペリフェラルを処理することを意図しています。

表 A3-8 メモリ属性の要約

メモリタイプ属性	共有可否	その他の属性	説明
ノーマル	共有可	キャッシュについて、次のいずれかの属性を持ちます。[a] キャッシュ不可 ライトスルーキャッシュ可 ライトバック書き込み割り当てキャッシュ可 ライトバック書き込み割り当てなしキャッシュ可	複数のプロセッサ間で共有されるノーマルメモリを処理することを意図しています。
	共有不可	キャッシュについて、次のいずれかの属性を持ちます。[a] キャッシュ不可 ライトスルーキャッシュ可 ライトバック書き込み割り当てキャッシュ可 ライトバック書き込み割り当てなしキャッシュ可	単一プロセッサによってのみ使用されるノーマルメモリを処理することを意図しています。

a. キャッシュ可属性は、内部および外部キャッシュ領域に対して独立して定義されます。

A3.5.3 ARM アーキテクチャにおけるアトミック性

*アトミック性*は、*アトミック*アトミックアクセスと記述される、メモリアクセスの一機能です。ARM アーキテクチャの説明では、2 種類のアトミック性が使用されます。定義については、次を参照して下さい。

- *シングルコピーのアトミック性*
- *マルチコピーのアトミック性* : P.A3-22

シングルコピーのアトミック性

次の条件がいずれも True の場合、読み出し / 書き込み操作は*シングルコピーアトミック*です。

- あるひとつのオペランドに対して任意の回数の書き込みの後、そのオペランドの値はその書込み操作のうちのひとつで書き込まれたものである。オペランドの値の一部はある書き込みから来て、値の別の一部はまた異なる書き込みからくるということはあり得ない。
- 同じオペランドに対して読み出し操作と書き込み操作が実行される場合、読み出し操作によって得られる値が、次のいずれかである。
  - — 書き込み操作前のオペランドの値
  - — 書き込み操作後のオペランドの値

  読み出された値の一部は書き込み操作前のオペランドの値であり、別の一部は書き込み操作後のオペランドの値であるということは決して発生しない。

ARMv7-M では、プロセッサによる次のアクセスがシングルコピーアトミックです。

- すべてのバイトアクセス
- ハーフワードアラインドの位置に対するすべてのハーフワードアクセス
- ワードアラインドの位置に対するすべてのワードアクセス

LDM、LDC、LDC2、LDRD、STM、STC、STC2、STRD、PUSH、POP 命令は、ワードアラインドのワードアクセスのシーケンスとして実行されます。各 32 ビット ワードアクセスは、シングルコピーアトミックであることが保証されます。このシーケンスに含まれる 2 つ以上のワードアクセスによるサブシーケンスは、シングルコピーのアトミック性を示さない可能性があります。

アクセスがシングルコピーアトミックでない場合は、より小さいアクセスのシーケンスとして実行され、各シーケンスは少なくともバイトレベルではシングルコピーアトミックです。

命令がこれらの規則に従うアクセスのシーケンスとして実行される場合、シーケンス内でいくつかの例外処理を行うことが可能で、命令の実行が放棄されることがあります。

例外復帰時には、アクセスのシーケンスを生成した命令が再実行されるため、例外を処理する前にすでに実行されたすべてのアクセスが繰り返し実行される可能性があります。

注

これらの複数アクセス命令に対する例外動作のため、これらの命令はソフトウェア同期化を目的としたメモリへの書き込みには適していません。

暗黙アクセスについては、次の規則が適用されます。

- キャッシュのラインフィルおよび退出は、明示的なトランザクションや命令フェッチのシングルコピーのアトミック性に影響を与えません。
- 命令フェッチは、フェッチされた各命令に対してシングルコピーアトミックです。

  注

  32 ビット Thumb 命令は、2 つの 16 ビット項目としてフェッチされます。

マルチコピーのアトミック性

マルチプロセッシングシステムでは、次の条件がいずれも True の場合、メモリ位置への書き込みはマルチコピーアトミックです。

- 同じ位置への書き込みはすべて*直列化*される。これは、すべてのオブザーバから見て、書き込みが同じ順序で観測されることを意味します。ただし、一部のオブザーバがすべての書き込みを観測しない可能性はあります。
- すべてのオブザーバによって書き込みが観測されるまで、ある位置からの読み出しがその書き込みの値を返すことはない。

ノーマルメモリへの書き込みはマルチコピーアトミックではありません。

デバイスおよびストロングリオーダメモリへのシングルコピーアトミックであるすべての書き込みは、マルチコピーアトミックでもあります。

同じ位置への書き込みアクセスはすべて直列化されます。ノーマルメモリへの書き込みアクセスは、同じアドレスへの別の書き込みが観測されるポイントまで繰り返すことが可能です。

ノーマルメモリの場合は、書き込みの直列化により書き込みのマージが禁止されることはありません。

A3.5.4 ノーマルメモリ

ノーマルメモリは「等べき」です。つまり、次の属性を示します。

- 読み出しアクセスは副作用なしに繰り返し可能である。
- 繰り返された読み出しアクセスは、読み出されたリソースに最後に書き込まれた値が返される。
- 読み出しアクセスは、副作用なしに追加のメモリ位置をプリフェッチ可能である。
- 書き込みアクセスは、書き込みを繰り返す間にその位置の内容が変化しない限り、副作用なしに繰り返し可能である。
- アンアラインドアクセスをサポート可能である。
- ターゲット メモリシステムへのアクセス前にアクセスをマージ可能である。

ノーマルメモリは、読み出し / 書き込み可または読み出し専用であり、ノーマルメモリ領域は共有可または共有不可のいずれかとして定義されます。

ノーマルメモリのタイプ属性は、システムで使用される大部分のメモリに適用されます。

ノーマルメモリへのアクセスは、メモリオーダリングの弱い一貫性モデルを持ちます。弱い一貫性メモリモデルの説明については、メモリオーダリングについての標準的な解説、例えば『*Memory Consistency Models for Shared Memory-Multiprocessors*』(Kourosh Gharachorloo, Stanford University Technical Report CSL-TR-95-685) の第 2 章を参照して下さい。一般にノーマルメモリでは、他のオブザーバから見たメモリアクセスの順序を制御する必要がある場合、バリア操作を使用することが必要になります。この要件は、ノーマルメモリ領域のキャッシュ可および共有可属性にかかわらず適用されます。

P.A3-33「*メモリアクセスの順序付けの要件*」で説明されているアクセスのオーダリング要件は、すべての明示的なアクセスに適用されます。

P.A3-21「*ARM アーキテクチャにおけるアトミック性*」で説明されているアクセスのシーケンスを生成する命令は、アクセスのシーケンス内で実行した例外処理の結果として、破棄される場合があります。この命令は例外からの復帰時に再始動され、それによって 1 つ以上のメモリ位置が複数回アクセスされることがあります。この場合、書き込みアクセス間で変更された位置への書き込みアクセスが繰り返されます。

共有不可のノーマルメモリ

ノーマルメモリ領域に対して、共有不可属性は、そのノーマルメモリがひとつのプロセッサだけにアクセスされることになるノーマルメモリであることを指定します。

共有不可でノーマルとして指定されたメモリ領域には、データおよび命令アクセスに対してキャッシュの影響を透過的にするための要件はありません。他のオブザーバによってメモリシステムが共有されており、キャッシュの存在がオブザーバ間の通信の際にコヒーレンシの問題を引き起こす可能性がある場合は、ソフトウェアはキャッシュ保守操作を使用する必要があります。このキャッシュ保守要件は、メモリオーダリングを保証するために必要なバリア操作に追加されます。

共有不可のノーマルメモリに対して、排他ロード / 排他ストアの同期化基本命令では、複数のオブザーバによるアクセスの可能性は考慮されません。

共有可のノーマルメモリ

ノーマルメモリに対して、共有可のメモリ属性は、複数のプロセッサまたは他のシステムマスタによってアクセスされる可能性のあるノーマルメモリであることを指定します。

共有可属性を持つノーマルメモリの領域は、メモリシステム上のキャッシュの挿入の影響が、同じ共有可のドメイン内でのデータアクセスに対して完全に透過的である領域です。命令キャッシュのコヒーレンシを保証するには、明示的なソフトウェア管理が必要になります。

実装では、この管理要件をサポートするために、共有可領域へのアクセスをキャッシュしない単純な方法や、それらの領域についてキャッシュのコヒーレンシを維持するためのより複雑なハードウェア方式を使用する方法など、各種の機構を使用できます。

共有可なノーマルメモリに対して、排他ロード / 排他ストアの同期化基本命令では、同じ共有可なドメインに対して複数のオブザーバによるアクセスの可能性が考慮されます。

注

共有可という概念を使用すると、システム設計者はノーマルメモリ内でコヒーレンシ要件を持つ必要のある位置を指定できます。ただし、ソフトウェアの移植を容易にするためには、ソフトウェアがあるメモリ領域を共有不可と指定することで共有メモリシステムでの異なるプロセッサ間でメモリにコンシステンシがないものとして処理することを、ソフトウェア開発者は仮定すべきではありません。このような想定は、共有可の概念を使用している異なるマルチプロッセシング実装間では移植不能です。どのマルチプロセッシング実装も、異なるプロセッシングエレメント間で本質的に共有されるキャッシュが実装されている可能性があります。

ライトスルーキャッシュ可、ライトバックキャッシュ可、およびキャッシュ不可のノーマルメモリ

ノーマルメモリの各領域には、共有可または共有不可の他に、次のうちのいずれかとしてマークされます。

- ライトスルーキャッシュ可
- ライトバックキャッシュ可。この場合、追加の修飾子によってさらに次のいずれかにマークされます。
  - ライトバック、書き込み割り当て
  - ライトバック、書き込み割り当てなし
- キャッシュ不可

領域に対するキャッシュ可属性は、その領域の共有可属性とは独立しています。キャッシュ可属性は、共有データの処理以外の目的に使用される場合、データ領域の処理が必要になることを示します。この独立関係は、例えば、キャッシュ可で共有可にマークされているメモリ領域であっても、共有可な領域のデータをキャッシュしない実装では、キャッシュに保持されない可能性があることを意味します。

A3.5.5 デバイスメモリ

デバイスメモリのタイプ属性は、メモリ位置へのアクセスが副作用を引き起こす、またはロードに対して返される値が実行されたロードの回数に応じて変化するようなメモリ位置を示します。メモリマップされたペリフェラルおよび I/O 位置は、通常デバイスとしてマークされるメモリ領域の例です。

デバイスとしてマークされているメモリに対してプロセッサから明示的なアクセスを行う場合、次の規則が適用されます。

- すべてのアクセスは、プログラムに指定されたサイズで行われます。
- アクセスの回数は、プログラムによって指定された回数と等しくなります。

プログラムが一回しかアクセスしていなければ、実装は対応するデバイスメモリ位置へのアクセスを繰り返してはいけません。言い換えれば、デバイスメモリ位置へのアクセスは再始動可能ではありません。

アーキテクチャでは、デバイスとしてマークされたメモリへの投機的アクセスは許可されていません。

デバイスとしてマークされたアドレス位置はキャッシュ不可です。デバイスメモリへの書き込みをバッファすることはできますが、書き込みのマージは、次の条件が維持される場合のみ行えます。

- アクセスの回数
- アクセスの順序
- 各アクセスのサイズ

同じアドレスへ複数のアクセスが行われる場合、そのアドレスに対するアクセスの回数が変化しないことが必要です。アクセスの結合は、デバイスメモリへのアクセスでは許可されていません。

デバイスメモリの操作に、ノーマルメモリ領域に影響を与える副作用がある場合は、ソフトウェアでメモリバリアを使用して正しい実行を保証する必要があります。例としては、メモリコントローラの構成レジスタをプログラミングする場合の、そのコントローラによって制御されるメモリとの関係が挙げられます。

デバイスメモリへのすべての明示的アクセスは、P.A3-33「*メモリアクセスの順序付けの要件*」で説明されているアクセスの順序付け要件に従う必要があります。

P.A3-21「*ARM アーキテクチャにおけるアトミック性*」で説明されているアクセスのシーケンスを生成する命令は、アクセスのシーケンス中に処理された例外の結果として破棄されることがあります。例外からの復帰時に命令が再始動されるため、1 つ以上のメモリ位置が複数回アクセスされることがあります。この場合、書き込みアクセス間で変更された位置への書き込みアクセスが繰り返される可能性があります。

注

命令が例外の後で再始動され、書き込みアクセスを繰り返す可能性がある場合は、デバイスメモリにアクセスするアクセスのシーケンスを生成する命令は使用しないで下さい。

アライメント制約事項によってフォルトとされていないアンアラインドアクセスがデバイスメ

モリに対して行われた場合、動作は予測不能です。

デバイスメモリ領域に対する共有可属性

デバイスメモリ領域には、共有可属性を与えることができます。これは、デバイスメモリの領域に次のいずれかを指定できることを意味します。

- 共有可のデバイスメモリ
- 共有不可のデバイスメモリ

共有不可のデバイスメモリは、単一プロセッサによるアクセスのみが可能として定義されます。共有可および共有不可のデバイスメモリをサポートするシステムの例として、次の両方をサポートする実装が挙げられます。

- 専用ペリフェラル用のローカルバス
- 共有されるメインのシステムバス上に実装されたシステムペリフェラル

このようなシステムでは、ウォッチドッグタイマまたは割り込みコントローラなどのローカルペリフェラルに対するアクセスタイムが予測しやすい可能性があります。特に、共有不可デバイスメモリ領域の特定のアドレスは、プロセッサごとに別の物理ペリフェラルにアクセスする可能性があります。

A3.5.6 ストロングリオーダメモリ

ストロングリオーダメモリのタイプ属性を持つメモリ領域には、プロセッサからのすべての明示的なメモリアクセスに対する強力なメモリオーダリングモデルが存在します。ストロングリオーダ属性を持つメモリへのアクセスは、DMB UN 命令がプロセッサからのアクセスの前後に挿入されている場合と同様に動作する必要があります。詳細については、P.A3-36「*データ メモリバリア (DMB)*」を参照して下さい。

同期化が必要な場合は、プログラムでメモリアクセスと次の命令との間に明示的なメモリバリアを含める必要があります。詳細については、P.A3-37「*データ同期化バリア (DSB)*」を参照して下さい。

プロセッサから、ストロングリオーダとしてマークされているメモリへ明示的なアクセスを行う場合、次の規則が適用されます。

- すべてのアクセスはプログラムで指定されたサイズで発生します。
- アクセスの回数は、プログラムによって指定された回数と等しくなります。

実装では、プログラムがストロングリオーダのメモリ位置へ 1 回しかアクセスしない場合、その位置へのアクセスを繰り返してはいけません。言い換えれば、ストロングオーダメモリ位置へのアクセスは再始動可能ではありません。

アーキテクチャでは、ストロングリオーダとしてマークされたメモリへの投機的アクセスは許可されていません。

ストロングリオーダメモリのアドレス位置はキャッシュに保持されず、常に共有可メモリ位置として扱われます。

ストロングリオーダメモリへのすべての明示的アクセスは、P.A3-33「*メモリアクセスの順序付けの要件*」で説明されているアクセスのオーダリング要件に適合している必要があります。

P.A3-21「*ARM アーキテクチャにおけるアトミック性*」で説明されているアクセスのシーケンスを生成する命令は、アクセスのシーケンス中に処理した例外の結果として破棄されることがあり

ます。例外からの復帰時に命令が再始動され、1 つ以上のメモリ位置が複数回アクセスされることがあります。この場合、書き込みアクセス間で変更された位置への書き込みアクセスが繰り返される可能性があります。

注

命令が例外の後に再始動されて、書き込みアクセスを繰り返す可能性がある場合は、ストロングリオーダメモリへのアクセスのシーケンスを生成する命令は使用しないで下さい。

アライメント制約事項によってフォルトとされていないアンアラインドアクセスがストロングリオーダメモリに対して行われた場合、動作は予測不能です。

A3.5.7 メモリアクセスの制約事項

メモリアクセスには、次の制約事項が適用されます。

- 任意のアクセス X について、X によってアクセスされるバイトはすべて同じメモリタイプ属性を持つ必要があり、そうでない場合アクセスの動作は予測不能になります。つまり、異なるメモリタイプ間の境界をまたぐアンアラインドアクセスは予測不能です。
- 同じ命令によって生成された 2 つのメモリアクセス X および Y について、X および Y によってアクセスされるバイトはすべて同じメモリタイプ属性を持つ必要があり、そうでない場合結果は予測不能です。例えば、ノーマルメモリとデバイスメモリとの間をまたぐ LDC、LDM、LDRD、STC、STM、STRD 命令は予測不能です。
- デバイスメモリまたはストロングリオーダメモリへのアンアラインド メモリアクセスを生成する命令は、予測不能です。
- デバイスメモリまたはストロングリオーダメモリへのアクセスを生成する命令の場合、命令の擬似コードによって指定されるアクセスのシーケンスを実装で変更することはできません。この条件には、次の内容を変更しないことが含まれます。
  - — アクセスの回数
  - — アクセスの時間的な順序
  - — 各アクセスのデータサイズおよび他のプロパティ

  さらに、プロセッサコアの実装では、接続されている任意のメモリシステムがアクセスのメモリタイプを識別し、アクセスの回数、時間的な順序、データサイズ、および他のプロパティに関する同様の制約事項に従うことを期待しています。

  この規則の例外は、次のとおりです。

  — あるプロセッサコアの実装では、メモリシステムに提供される情報を使って、アクセスの元の回数、時間的な順序、および他の詳細を再構築することが可能であれば、この規則を守る必要はありません。さらに、その実装ではアクセスがデバイスメモリまたはストロングリオーダメモリに対して行われる場合、接続されているメモリシステムがこの再構築を実行することを要件とする必要があります。

  例えば、64 ビットバスを使用する実装では、LDM 命令によって生成されたワードロードを 64 ビットアクセスにペアリングできます。これは、命令セマンティクスにより、64 ビットアクセスでは常に下位アドレスからのワードロードに続いて上位アドレスからのワードロードの順序で実行されることが保証されているためで

す。ただし、実装ではアクセスがデバイスメモリまたはストロングリオーダメモリに対して行われる場合に、メモリシステムが 2 つのワードロードをアンパックすることが許可される必要があります。

— どのような場合でも、上で説明されたものと異なる結果が観測されなければ、任意の実装は正当なものです。

- LDM および STM 命令は IT 命令とともに使用され、実行中に割り込みが発生した場合は再始動可能です。ロード / ストア命令の再始動は、デバイスメモリおよびストロングリオーダメモリのアクセス規則と互換性はありません。
- PC をロード / ストアするマルチアクセス命令では、ノーマルメモリのみにアクセスできます。デバイスメモリまたはストロングリオーダメモリへのアクセス命令が実行された場合、結果は予測不能です。
- 命令フェッチでは、ノーマルメモリのみにアクセスできます。デバイスメモリまたはストロングリオーダメモリへのアクセスが実行された場合、結果は予測不能です。例えば、命令フェッチと明示的アクセスとの間には順序付け要件が存在しないため、読み出しで影響を受けるデバイスを含むメモリ領域に対して命令フェッチを実行することはできません。

正確さを保証するには、読み出しで影響を受ける位置が実行不可としてマークされている必要があります（P.A3-29「*命令アクセスの特権レベルアクセス制御*」参照）。

A3.6 アクセス権

ARMv7 にはメモリ領域に対する追加の属性が組み込まれており、次の処理が可能になります。

- アクセスの特権に基づいて、データアクセスを制限できます。詳細については、*データアクセスの特権レベルアクセス制御*を参照して下さい。
- フェッチを行うプロセスまたはスレッドの特権に基づいて、命令フェッチを制限できます。詳細については、*命令アクセスの特権レベルアクセス制御*を参照して下さい。

A3.6.1 データアクセスの特権レベルアクセス制御

メモリ属性によって、メモリ領域に次のアクセス条件を定義できます。

- どのようなアクセスも行えない。
- 特権アクセスのみ行える。
- 特権アクセスおよび非特権アクセスの両方が行える。

アクセスの特権レベルは、明示的な読み出しおよび書き込みアクセスに対して個別に定義されます。ただし、メモリ属性を定義するシステムでは、読み出しおよび書き込みアクセスに対するメモリ属性のすべての組み合わせをサポートする必要はありません。

特権アクセスは特権モードでの実行中に、LDRT、STRT、LDRBT、STRBT、LDRHT、STRHT、LDRSHT、LDRSBT 命令を除くロード / ストア操作の結果として行われるアクセスです。

非特権アクセスは、次のいずれかの場合に、ロード / ストア操作の結果として行われるアクセスです。

- 現在の実行モードが非特権アクセス専用に構成されている。
- プロセッサのモードにかかわらず、LDRT、STRT、LDRBT、STRBT、LDRHT、STRHT、LDRSHT、LDRSBT のいずれかの命令の結果としてアクセスが行われる。

プロセッサが、アクセス権によって許可されていないデータアクセスを試みた場合は、例外が発生します。例えば、プロセッサが非特権モードのとき、特権アクセスのみ可能としてマークされているメモリ領域にアクセスを試みた場合は、MemManage 例外が発生します。

注

データアクセス制御は、*メモリ保護ユニット*が実装され、有効な場合にのみサポートされます。詳細については、P.B2-12「*保護メモリシステム アーキテクチャ*」を参照して下さい。

A3.6.2 命令アクセスの特権レベルアクセス制御

メモリ属性によって、メモリ領域について次のアクセス条件を定義できます。

- 実行アクセスは不可
- 特権プロセスのみによって、実行アクセスが可能
- 特権および非特権のプロセスによって、実行アクセスが可能

メモリ領域への命令アクセス権を定義するため、メモリ属性には領域に対してそれぞれ独立して、次の条件が定義されます。

- 読み出しアクセス権。詳細については、P.A3-29「*データアクセスの特権レベルアクセス制御*」を参照して下さい。
- *実行に適しているか*どうか。

例えば、特権プロセスによってのみ実行アクセスが可能な領域には、次のメモリ属性が指定されています。

- 特権読み出しアクセスのみでアクセスが可能
- 実行に適している。

これは、明示的なメモリアクセスについて、領域へのアクセス可能性を定義するメモリ属性と、領域が実行可能であることを定義するメモリ属性との間に、何らかの関係があることを意味しています。

コード実行が許可されていない属性を持つメモリ位置からプロセッサがコードの実行を試みると、MemManage 例外が発生します。

注

命令アクセス制御は、メモリ保護ユニットが実装され、有効な場合に完全にサポートされます。詳細については、P.B2-12「*保護メモリシステム アーキテクチャ*」を参照して下さい。

命令実行アクセス制御は、デフォルトのアドレスマップでもサポートされています。詳細については、P.B2-2「*システム アドレスマップ*」を参照して下さい。

A3.7 メモリアクセスの順序

ARMv7 では、明確なメモリアクセス プロパティを持つ、ノーマル、デバイス、ストロングリオーダの 3 つのメモリタイプのセットが用意されています。

メモリ属性の ARMv7 アプリケーションレベルのビューについては、以下で説明されています。

- *メモリのタイプおよび属性とメモリオーダ モデル* : P.A3-19
- *アクセス権* : P.A3-29

メモリアクセスの順序付けについて考える場合、重要な機能は*共有可*メモリ属性です。この属性は、メモリの領域が複数のプロセッサ間で共有可か、およびその結果として順序モデルで外見上のキャッシュの透過性が必要かどうかを示します。

メモリ順序モデルに関する主な考慮点は、対象読者によって次のように異なります。

- ソフトウェアプログラマの場合は、アプリケーションレベルでモデルを考えるため、主な要素はノーマルメモリに対するアクセスで、他のオブザーバによって観測されるアクセスの順序を制御することが必要とされる一部の状況においては、バリアが必要になります。
- シリコン実装者の場合は、システムレベルでモデルを考えるため、ストロングオーダメモリおよびデバイスメモリ属性により、どのようなものが構築可能か、およびどの時点でアクセスの完了を示すかについて、システム設計者に一定の制限が課されます。

  注

  メモリモデルの機能を実装するのに必要な機構の選択は、実装者に任されています

メモリ順序モデルの詳細については、次のサブセクションを参照して下さい。

- *読み出しと書き込み* : P.A3-31
- *メモリアクセスの順序付けの要件* : P.A3-33
- *メモリバリア* : P.A3-36

メモリシステム アーキテクチャに関連する追加の属性および動作については、P.B2-12「*保護メモリシステム アーキテクチャ*」を参照して下さい。

A3.7.1 読み出しと書き込み

メモリへの各アクセスは、読み出しまたは書き込みのいずれかです。*明示的*メモリアクセスは、命令の機能によって要求されるメモリアクセスです。次の操作は、明示的でないメモリアクセスを発生する可能性があります。

- 命令フェッチ
- キャッシュのロードおよびライトバック

特記されていない限り、メモリオーダリング要件は明示的なメモリアクセスのみに適用されます。

読み出し

読み出しは、ロードの意味を持つメモリ操作として定義されます。

次の命令のメモリアクセスは、読み出しです。

- LDR、LDRB、LDRH、LDRSB、LDRSH
- LDRT、LDRBT、LDRHT、LDRSBT、LDRSHT
- LDREX、LDREXB、LDREXH
- LDM{IA,DB}、LDRD、POP
- LDC および LDC2
- STREX、STREXB、STREXH によって返されるステータス値
- TBB および TBH

書き込み

書き込みは、ストアの意味を持つメモリ操作として定義されます。

次の命令のメモリアクセスは、書き込みです。

- STR、STRB、STRH
- STRT、STRBT、STRHT
- STREX、STREXB、STREXH
- STM{IA,DB}、STRD、PUSH
- STC および STC2

同期化基本命令

同期化基本命令では、メモリ順序モデルにおいてシステムセマフォの正しい操作を保証する必要があります。次の同期化基本命令は、メモリの同期化を保証するために使用される命令として定義されます。

- LDREX、STREX、LDREXB、STREXB、LDREXH、STREXH

排他ロード、排他ストア、排他クリア命令の詳細については、P.A3-9「*同期化とセマフォ*」を参照して下さい。

排他ロードおよび排他ストア命令は、共有可メモリおよび共有不可メモリに対してサポートされています。共有不可メモリは、同じプロセッサ上で実行中のプロセスの同期に使用できます。プロセスが異なるプロセッサで実行されている可能性がある場合、プロセスの同期には共有可を使用する必要があります。

観測可能性と完了

メモリアクセスを観測できるオブザーバのセットは、システムによって定義されます。
すべてのメモリについて、次の規則が適用されます。

- メモリのある位置への書き込みは、オブザーバがそのメモリ位置を次に読み出したときに、その書き込みによって書き込まれた値が返される場合、そのオブザーバによって観測されると呼びます。

- メモリのある位置への書き込みは、そのメモリが共有可ドメインに含まれていて、書き込みを観測可能な任意のオブザーバが、次にそのメモリ位置を読み出したときに、書き込まれた値が返される場合、共有可なドメインに対してグローバルに観測されると呼びます。
- メモリのある位置からの読み出しは、オブザーバがそのメモリ位置へ次に行う書き込みが、読み出される値に影響しない場合、そのオブザーバによって観測されると呼びます。
- メモリのある位置からの読み出しは、そのメモリが共有可ドメインに含まれていて、書き込みを観測可能な任意のオブザーバがそのメモリ位置へ次に行う書き込みが、読み出される値に影響しない場合、共有可ドメインについてグローバルに観測されると呼びます。

さらに、ストロングリオーダメモリの場合は次の規則が適用されます。

- 副作用を示すペリフェラル内のメモリマップされた位置に対する読み出し / 書き込みは、その読み出し / 書き込みが次の条件を満たす場合にのみ観測、またはグローバルに観測されると呼びます。
  - リストされている一般的な条件を満たしている。
  - メモリマップされたペリフェラルの状態への影響を与え始める可能性がある。
  - 他のペリフェラルデバイス、コア、またはメモリに対して、すべての関連する副作用をトリガする可能性がある。

すべてのメモリについて、ARMv7-M の完了規則は次のように定義されます。

- 次のすべての条件が True の場合に、共有可ドメインに対する読み出し / 書き込みが完了します。
  - その共有可ドメインについて、読み出し / 書き込みがグローバルに観測される。
  - 共有可ドメイン内のオブザーバによるすべての命令フェッチが、読み出し / 書き込みをすでに観測している。
- キャッシュまたは分岐予測器の保守操作は、その操作の影響が共有可ドメイン内でグローバルに観測された場合、その共有可ドメインに対して完了します。

ストロングリオーダメモリおよびデバイスメモリでの副作用の完了

ストロングリオーダメモリまたはデバイスメモリ内でのメモリアクセスが完了しても、メモリアクセスの副作用がすべてのオブザーバに対して可視であることは保証されません。メモリアクセスの副作用の可視性を保証する機構は実装定義で、例えばポーリング可能なステータスレジスタの搭載がそれに相当します。

A3.7.2 メモリアクセスの順序付けの要件

ARMv7-M では、メモリアクセスの許可された順序付けにおけるアクセスの制約事項が定義されています。これらの制約事項は、該当するアクセスのメモリ属性に依存します。

メモリアクセスの順序付けの要件を説明するために、次の 2 つの用語が使用されます。

アドレス依存関係

アドレス依存関係は、読み出しアクセスによって返された値が次の読み出し / 書き込みアクセスのアドレスを計算するために使用される場合に存在します。最初の読み出しアクセスによって読み出された値が 2 番目の読み出し / 書き込みアクセスのアドレスを変更しない場合でも、アドレス依存関係は存在します。これは、返された値が使用される前にマスクオフされるか、または 2 番目のアクセスの予測されたアドレス値に影響を与えない場合です。

制御依存関係

制御依存関係は、読み出しアクセスによって返されたデータ値が条件コードフラグを決定するために使用され、そのフラグの値を条件コードの評価に使用して次の読み出しアクセスのアドレスが決定される場合に存在します。このアドレス決定は、条件付き実行または分岐の評価を通して行われます。

P.A3-35 図 A3-4 は、2 回の明示的アクセス A1 と A2 のメモリ順序を示したものです (A1 はプログラムで A2 よりも前にあります)。この図では、次の記号を使用しています。

< アクセスはプログラムの順序でグローバルに観測される必要があります。つまり、A1は厳密にA2より前のアクセスとしてグローバルに観測される必要があります。

\- ユニプロセッサセマンティクスの要件が満たされている場合、例えば単一プロセッサ内において命令間の依存関係が順守されている場合、アクセスはどのような順序でグローバルに観測されてもかまいません。

次の追加の制約事項は、この記号を持つメモリアクセスの順序付けに適用されます。

- アドレス依存関係が存在する場合、2 つのメモリアクセスはプログラムの順序で観測されます。

  この順序付けの制約事項は、2 つの読み出しアクセスの間に制御依存関係のみが存在する場合には適用されません。
  2つの読み出しアクセス間にアドレス依存関係および制御依存関係の両方が存在する場合は、アドレス依存関係の順序付け要件が適用されます。
- 読み出しアクセスによって返された値が次の書き込みアクセスによって書き込まれるデータとして使用される場合、2 つのメモリアクセスはプログラムの順序で観測されます。
- メモリ位置にプログラムのシーケンシャルな実行で書き込みが行われない場合、オブザーバがそのメモリ位置への書き込みアクセスを観測することはできません。
- 値がプログラムのシーケンシャルな実行で書き込まれない場合、オブザーバがそのメモリ位置への書き込み値を観測することはできません。

P.A3-35 図 A3-4 で、アクセスとは指定されたメモリタイプへの読み出し / 書き込みアクセスを意味しています。例えば、*デバイスアクセス*、*共有不可*は、共有不可デバイスメモリへの読み出し / 書き込みアクセスの意味です。

A1 \ A2	通常アクセス	デバイスアクセス		ストロングオーダアクセス
		共有不可	共有可	
通常アクセス	-	-	-	<
デバイスアクセス、共有不可	-	<	-	<
デバイスアクセス、共有可	-	-	<	<
ストロングオーダ アクセス	<	<	<	<

図 A3-4 メモリ順序の制約事項

どのタイプのメモリの場合でも、暗黙アクセスの順序付け要件は存在しません。

命令実行のプログラム順序

命令実行のプログラム順序は、制御フロートレース内の命令の順序になります。

実行における明示的メモリアクセスは、次のいずれかです。

厳密な順序付け
: < で示されます。厳密に指定の順序で行われる必要があります。

順序付け
: <= で示されます。指定の順序で、または同時に行うことができます。

複数ロード / ストア命令、LDC、LDC2、LDMDB、LDMIA、LDRD、POP、PUSH、STC、STC2、STMDB、STMIA、STRD 命令は複数のワードアクセスを生成し、順序付け決定に関してはそれぞれが別のアクセスとして扱われます。

2 つのアクセス A1 と A2 について、プログラム順序を決定するための規則を以下に示します。

A1 と A2 が 2 つの異なる命令により生成されている場合

- プログラム順序で A1 を生成する命令が A2 を生成する命令よりも前にある場合、A1 < A2
- プログラム順序で A2 を生成する命令が A1 を生成する命令よりも前にある場合、A2 < A1

A1 と A2 が同じ命令により生成されている場合

- A1 と A2 が LDC、LDC2、LDMDB、LDMIA、POP 命令によって生成される 2 つのワードロードであるか、または PUSH、STC、STC2、STMDB、STMIA 命令によって生成される 2 つのワードストアの場合。PC を含むレジスタリストを持つ LDMDB、LDMIA、POP 命令は除外されます。
  - A1 のアドレスが A2 のアドレスよりも小さい場合、A1 <= A2
  - A2 のアドレスが A1 のアドレスよりも小さい場合、A2 <= A1
- A1 と A2 が、PC を含むレジスタリストを持つ LDMDB、LDMIA、POP 命令によって生成される 2 つのワードロードである場合、メモリアクセスのプログラム順序は定義されません。

- A1 と A2 が、LDRD 命令によって生成される 2 つのワードロード、または STRD 命令によって生成される 2 つのワードストアである場合、メモリアクセスのプログラム順序は定義されません。
- このセクションで明示的に説明されていない命令または操作については、P.A3-21「*シングルコピーのアトミック性*」で説明されているシングルコピーのアトミック性の規則により操作がアクセスのシーケンスになることを意味する場合、それらのアクセスの時間的な順序は定義されません。

A3.7.3 メモリバリア

*メモリバリア*は、命令または命令のシーケンスに適用される一般的な用語で、プロセッサコア内でロード / ストア命令の終了に関して、プロセッサによる同期イベントを強制的に発生させるために使用されます。メモリバリアは、次の両方を保証するのに使用されます。

- プログラマモデルに対して、前のロード / ストア命令が完了したこと。
- メモリバリアイベント前にプリフェッチされたすべての命令がフラッシュされたこと。

ARMv7-M では、本章で説明されているメモリ順序モデルをサポートするため、次に示す 3 つの明示的なメモリバリアが必要です。

- データメモリバリア。詳細については、*データ メモリバリア (DMB)* を参照して下さい。
- データ同期化バリア。詳細については、P.A3-37「*データ同期化バリア (DSB)*」を参照して下さい。
- 命令同期化バリア。詳細については、P.A3-38「*命令同期化バリア (ISB)*」を参照して下さい。

DMB および DSB メモリバリアは、ロード / ストア命令によって生成されるメモリシステムへの読み出し / 書き込みに影響を与えます。命令フェッチは明示的なアクセスではないため、影響を受けません。

注

ARMv7-M では、メモリバリア操作をデータまたは統一キャッシュおよび分岐予測の保守操作と組み合わせて使用することが必要な場合があります。

データ メモリバリア (DMB)

DMB 命令は、データ メモリバリアです。DMB 命令を実行するプロセッサは、*実行中プロセッサ* Pe として示されます。DMB 命令は、共有可必須ドメインおよび*必要アクセスタイプ*を引数として使用します。

注

ARMv7-M では、共有不可ドメインまたはアクセスタイプ制限を持つシステム全体のバリアのみがサポートされています。

DMB では、*グループ A* および *グループ B* という 2 つのメモリアクセス グループが作成されます。

グループ A 次のものから構成されます。

- Pe と同じ共有可ドメイン内のオブザーバによって行われ、DMB 命令の前に Pe によって観測される、必要アクセスタイプのすべての明示的メモリアクセス。これには、Pe によって実行される必要アクセスタイプおよび共有可必須ドメインのすべてのアクセスが含まれます。

- Pe と同じ共有可ドメイン内のオブザーバによる、必要アクセスタイプのすべてのロードで、Pe と同じ共有可必須ドメイン内の任意のオブザーバ Py によって、Py がグループ A のメンバであるメモリアクセスを実行する前に観測されるもの

グループ B 次のものから構成されます。

- Pe による必要アクセスタイプのすべての明示的メモリアクセスで、プログラム順序で DMB 命令の後に発生するもの
- Pe と同じ共有可必須ドメイン内の任意のオブザーバ Px による、必要アクセスタイプのすべての明示的メモリアクセスで、Px がグループ B のメンバであるストアを観測した後にのみ発生可能なもの

Pe と同じ共有可必須ドメインを持つ任意のオブザーバは、グループ B のいずれかのメンバを観測する前にグループ A のすべてのメンバを観測します。グループ A およびグループ B のメンバは同じメモリマップされたペリフェラルにアクセスしますが、グループ A のすべてのメンバは、グループ B のいずれかのメンバがそのメモリマップされたペリフェラルから可視になる前に、そのペリフェラルから可視になります。

注

- メモリアクセスは、グループ A とグループ B のいずれにも属さない可能性があります。DMB は、そのようなメモリアクセスの観測の順序には影響を与えません。
- グループ A の定義の 2 番目の部分は再帰的です。最終的に、グループ A のメンバ構成は、グループ A の定義の最初の部分によりグループ A のメンバとして定義されるアクセスを Py が実行する前に、Py がロードを観測することから導き出されます。
- グループ B の定義の 2 番目の部分は再帰的です。最終的に、グループ B のメンバ構成は、グループ B の定義の最初の部分によりグループ B のメンバとして定義される、Pe によるアクセスが、任意のオブザーバによって観測されることから導き出されます。

DMB はメモリアクセスにのみ影響を与えます。プロセッサ上で実行されている他の命令の順序には影響を与えません。

DMB 命令の詳細については、P.A6-68「*DMB*」を参照して下さい。

データ同期化バリア (DSB)

DSB 命令は特別なメモリバリアで、実行ストリームをメモリアクセスと同期します。DSB 命令は、*共有可必須ドメイン*および*必要アクセスタイプ*を引数として使用します。DSB は同じ引数を持つ DMB として動作し、ここで定義されている追加のプロパティも持っています。

注

ARMv7-M では、共有不可ドメインまたはアクセスタイプ制限を持つシステム全体のバリアのみがサポートされています。

DSB 命令は、次の両方の条件が満たされた場合に完了します。

- DSB が実行される前に Pe によって観測され、必要アクセスタイプで、Pe と同じ共有可必須ドメイン内のオブザーバから発生したすべての明示的メモリアクセスが、その共有可必須ドメイン内のオブザーバのセットに対して完了する。
- DSB が完了する前に Pe によって発行された、すべてのキャッシュおよび分岐予測器の保守操作が完了する。

さらに、プログラム順序で DSB 命令の後に出現する命令は、DSB 命令が完了するまで実行されません。

DSB 命令の詳細については、P.A6-70「*DSB*」を参照して下さい。

命令同期化バリア (ISB)

ISB 命令は、プロセッサ内のパイプラインをフラッシュし、プログラム順序で ISB 命令の後に出現するすべての命令が、ISB 命令の完了後にのみキャッシュまたはメモリからフェッチされるようにします。ISB 命令を使用することにより、ISB 命令の前に実行されたコンテキスト変更操作の作用が、ISB 命令後にフェッチされた命令から可視であることが保証されます。操作が完了したことを保証するため、ISB 命令の挿入を必要とする可能性があるコンテキスト変更操作の例として、次のものが挙げられます。

- システム制御の更新が発生したことを保証する。
- 分岐予測器の保守操作

さらに、プログラム順序で ISB 命令の後に出現するすべての分岐命令は、ISB 命令後に可視になるコンテキストとともに分岐予測ロジックに書き込まれます。これは、命令ストリームの正確な実行を保証するために必要になります。

プログラム順序で ISB 命令の後に出現するすべてのコンテキスト変更操作は、ISB 命令が実行された後でのみ有効になります。

ARMv7-M の実装では、現在の実行ポイントからどれだけ先の命令をプリフェッチするかを選択する必要があります。この命令の数は、固定でも、動的に変化する数でもかまいません。プリフェッチする命令の数を選択するのと同様に、実行可能な以後の実行パスのうち、どのパスに沿ってプリフェッチするかを実装で選択することができます。例えば、分岐命令の後で、プログラム順序で分岐後に出現する命令か、または分岐ターゲットにある命令のいずれかをプリフェッチすることができます。これは分岐予測と呼ばれます。

すべての形式の命令プリフェッチにおける潜在的な問題は、プリフェッチしたメモリ内の命令が実行前に変更される可能性があるということです。これが発生した場合、メモリ内の命令が変更されても、プリフェッチ済みの命令のコピーが実行され完了することは通常は防止されません。必要であれば、メモリバリア命令の ISB、DMB、DSB を使用して、実行順序付けを強制的に行います。

ISB 命令の詳細については、P.A6-76「*ISB*」を参照して下さい。

A3.8 キャッシュとメモリ階層

ARMv7-M でのキャッシュのサポートは、メモリ属性に限られています。これらは、システムキャッシュをサポートするため AMBA（AHB または AXI プロトコル）などのサポートされているバスプロトコル上で出力されます。

コヒーレンシの破綻が発生する可能性がある状況では、ソフトウェアは、メモリマップされて実装定義であるキャッシュ保守操作を使用してキャッシュを管理する必要があります。

A3.8.1 キャッシュの概要

キャッシュは、アドレス情報（一般にタグと呼ばれます）および関連するデータの両方を含む高速なメモリ位置のブロックです。キャッシュの目的は、メモリアクセスの平均速度を上げることです。キャッシュは、次の 2 つの局所性の原則に基づいて動作します。

空間的局所性 ある位置がアクセスされた後で、隣接する位置がアクセスされる可能性が高いという原則です。例えば、シーケンシャルな命令実行やデータ構造の使用などの場合です。

時間的局所性 メモリ領域へのアクセスは、短時間内に繰り返される可能性が高いという原則です。例えば、繰り返しプログラムの実行の場合です。

格納される制御情報の数を最小限にするため、空間的局所性プロパティを使用して、同じタグのもとで複数の位置がグループ化されます。この論理ブロックは、一般にキャッシュラインと呼ばれます。データがキャッシュにロードされると、次のロード / ストアのアクセスタイムが短縮され、全体のパフォーマンスが向上します。すでにキャッシュにある情報へのアクセスはキャッシュヒットと呼ばれ、他のアクセスはキャッシュミスと呼ばれます。

通常、キャッシュは自己管理され、自動的に更新されます。プロセッサがキャッシュ可の位置にアクセスする場合、常にキャッシュがチェックされます。アクセスがキャッシュヒットの場合、アクセスは即座に発生し、そうでない場合は位置が割り当てられ、キャッシュラインがメモリからロードされます。別のキャッシュトポロジおよびアクセス方式も実行可能ですが、基礎となるアーキテクチャのメモリコヒーレンシ モデルに準拠している必要があります。

キャッシュによりいくつかの潜在的な問題が生じますが、主な原因は次のとおりです。

- プログラマが通常予期していない時期にメモリアクセスが発生する可能性がある。
- データ項目が、物理的に複数の場所に存在することになる。

A3.8.2 アプリケーションプログラマへのキャッシュの影響

キャッシュは通常はアプリケーションプログラマに対して透過的ですが、コヒーレンシの破綻が発生した場合、可視になることがあります。このようなブレークダウンは、次の場合に発生します。

- メモリ位置がシステム内の他のエージェントによって更新される。
- アプリケーションコードから実行されたメモリ更新を、システム内の他のエージェントに対して可視にする必要がある。

例として、次のような場合が挙げられます。

プロセッサのデータキャッシュ内に保持されているメモリ位置を読み出す DMA を使用するシステムにおいて、プロセッサがデータキャッシュに新規データを書き込んだ際にコヒーレンシの破綻が発生したが、DMA ではメモリ内に保持されている以前のデータが読み出される。

ハーバードアーキテクチャのキャッシュで、新しい命令データがデータキャッシュやメモリに書き込まれたが、命令キャッシュはまだ古い命令データを格納しているとき、コヒーレンシの破綻が起きます。

A3.8.3 キャッシュのプリロード

ARM アーキテクチャでは、メモリシステム ヒントとして PLD（データのプリロード）および PLI（命令のプリロード）が用意されており、ソフトウェアがどのメモリ位置の使用を予定しているかをハードウェアに伝達することができます。メモリシステムは、それらのヒントに対する応答として、そのメモリアクセスが発生する場合にアクセスを高速化するために必要なアクションを実行できます。これらのメモリシステム ヒントの作用は実装定義です。一般に、実装ではこの情報を使用して、データまたは命令の位置を通常のメモリより高速なアクセス速度を持つキャッシュに移動します。

プリロード命令はヒントなので、実装ではデバイスの機能的な動作に影響を与えない NOP として扱われます。この命令は例外を生成しませんが、メモリシステムの動作の結果、メモリアクセスによる不正確なフォルト（非同期例外）を生成することがあります。

第 A4 章
ARMv7-M 命令セット

本章では、ARMv7-M の Thumb 命令セットについて説明します。本章は以下のセクションから構成されています。

A4.1 命令セットについて

ARMv7-M は Thumb-2 バージョンの Thumb 命令セットをサポートしています。使用可能な機能の多くは、ARMv6T2 や他の ARMv7 プロファイルで Thumb 命令セットとともにサポートされている ARM 命令セットと同一です。本章では、ARMv7-M の Thumb 命令セットで使用できる機能と、Thumb 命令セットまたは ARM 命令セットのいずれかにアセンブルできる*統一アセンブラ言語* (UAL) について説明します。

Thumb 命令は 16 ビットまたは 32 ビットで、2 バイト境界にアラインされています。16 ビットと 32 ビットの命令は自由に混在できます。多くの一般的な操作は、16 ビット命令を使用した場合に最も効率的に実行されます。ただし、16 ビット命令には次のような制限があります。

- ほとんどの 16 ビット命令は、R0 〜 R7 の 8 つの汎用レジスタにしかアクセスできません。これらのレジスタを、下位レジスタと呼びます。一部の 16 ビット命令は、R8 〜 R15 の上位レジスタにアクセスできます。
- 2 つ以上の 16 ビット命令を必要とする多くの操作は、1 つの 32 ビット命令によってより効率的に実行できます。

ARMおよびThumb-2命令セットは自由に*インターワーキング*が可能なように設計されています。ただし、ARMv7-M は Thumb 命令のみをサポートしているため、ARMv7-M でのインターワーキング命令は Thumb 状態での実行のみを指定する必要があります。詳細については P.A4-2 「*ARMv7-M とインターワーキングのサポート*」を参照して下さい。

その他、次の章を参照して下さい。

- Thumb-2 命令セットのエンコードの詳細については、第 A5 章 *「Thumb® 命令セットのエンコード」*
- 命令の詳細な説明については、第 A6 章 *「Thumb 命令の詳細」*

A4.1.1 ARMv7-M とインターワーキングのサポート

Thumb インターワーキングは、*インターワーキングアドレス*のビット [0] として保持されています。インターワーキングアドレスは、BX、BLX あるいは PC をロードする LDR または LDM 命令中で使用されます。

ARMv7-M は Thumb 命令実行状態のみをサポートしているため、インターワーキング命令のアドレスビット [0] は 1 の必要があり、そうでない場合はフォルトが発生します。すべての命令はビット [0] を無視し、PC を更新するときにビット [31:1]: "0" を書き込みます。

PC を更新する 16 ビット命令は次のように動作します。

- ADD（レジスタ）および MOV（レジスタ）命令は、Thumb 状態内でインターワーキングなしの分岐を行います。

  注

  ADD（SP + レジスタ）16 ビット命令で Rd として PC を使用することは推奨されません。

- B または B<cond> 命令は、インターワーキングなしの分岐を行います。
- BLX（レジスタ）および BX 命令は、Rm の値によるインターワーキングを行います。
- POP 命令は、PC にロードされた値によるインターワーキングを行います。
- BKPT および SVC は例外を引き起こすため、インターワーキング命令とはみなされません。

PC を更新する 32 ビット命令は次のように動作します。

- B または B<cond> 命令はインターワーキングなしの分岐を行います。
- BL 命令は、PC へ書き込まれた値のビット [0] に関係なく、命令エンコードに基づいて Thumb 状態へ分岐します。
- LDM および LDR 命令は、PC へ書き込まれた値を使用したインターワーキングをサポートします。
- TBB および TBH 命令は、インターワーキングなしで分岐を行います。

詳細については、P.A2-11「*ARM コアレジスタ演算の擬似コードの詳細*」にある BXWritePC() 関数の説明を参照して下さい。

A4.1.2 条件付き実行

*条件付き実行*とは、APSR の N、Z、C、V フラグがその命令で指定された条件を満たす場合のみ、命令がプログラマモデルの動作、メモリ、コプロセッサにその命令の通常の影響を及ぼすことを意味します。フラグがこの条件を満たさない場合、命令は NOP として動作します。つまり、実行は次の命令へと通常に進み、例外処理をするかのチェックも行われますが、他の影響は何も及ぼしません。

ほとんどの Thumb 命令は無条件です。Thumb コードの条件付き実行は、次のいずれかの命令を使用して行われます。

- 16 ビットの条件付き分岐命令（分岐範囲 - 256 〜 + 254 バイト）。詳細については、P.A6-40「*B*」を参照して下さい。ARMv6T2 で Thumb-2 命令セットが追加されるまで、Thumb コードで条件付き実行を行うための他の機構はありませんでした。
- 32 ビットの条件付き分岐命令（分岐範囲約± 1MB）。詳細については、P.A6-40「*B*」を参照して下さい。
- 16 ビットの比較して 0 で分岐、および比較して非 0 で分岐命令（分岐範囲 + 4 〜 + 130 バイト）。詳細については、P.A6-52「*CBNZ, CBZ*」を参照して下さい。
- 次に続く最大 4 つの命令を条件付き実行とする、16 ビットの If-Then 命令。詳細については、P.A6-78「*IT*」を参照して下さい。IT 命令によって条件付き実行に変更される命令を、*IT ブロック*と呼びます。IT ブロック内の命令はすべて同じ条件付きで実行するか、または一部の命令をある条件で、他の命令をその逆の条件で実行することができます。

条件付き実行の詳細については、P.A6-8「*条件付き実行*」を参照して下さい。

A4.2 統一アセンブラ言語

本書では、ARM 統一アセンブラ言語 (UAL) を使用します。このアセンブラ言語構文は、ARM および Thumb 命令の標準的な形式を示すものです。

UAL は、各命令のニーモニックとオペランドの構文を記述します。また、命令とデータの項目にはラベルを付けることができます。ラベルに使用される構文や、どのようなアセンブラディレクティブやオプションが使用できるかは指定されていません。これらの詳細については、アセンブラのドキュメントを参照して下さい。

以前の ARM アセンブリ言語ニーモニックは、命令の詳細で示すようにシノニムとしてサポートされています。

注

以前の Thumb アセンブリ言語ニーモニックのほとんどはサポート*されていません*。詳細については、付録 B *「過去の命令のニーモニック」*を参照して下さい。

UAL には命令選択規則があり、それにより、複数の命令エンコードが目的の機能を提供できる場合にどのエンコードを選択するかが指定されます。例えば、ADD R0,R1,R2 命令には 16 ビットと 32 ビットの両方のエンコードが存在します。最も一般的な命令選択規則は、16 ビットエンコードと 32 ビットエンコードの両方が使用できる場合は、コード密度を最適化するために 16 ビットエンコードを選択するというものです。

通常の命令選択規則をオーバーライドし、特定のエンコードが選択されることを保証する構文オプションが存在します。コードの逆アセンブルを行うときに、生成されるアセンブリが元のコードを生成することを保証するようにするなどの、いくつかの他の状況においてこれらのオプションは有用です。

A4.2.1 条件付き命令

ARM 命令セットと Thumb-2 命令セットとの間で UAL アセンブリ言語の移植性を最大にするため、次の基準をお勧めします。

- Thumb-2 命令セットにおける正しい方法で、条件付き命令の前に IT 命令を記述します。
- ARM 命令セットへアセンブルするとき、アセンブラはすべての IT 命令が正しいことをチェックしますが、それらのコードは生成しません。

他の Thumb 命令は無条件ですが、IT 命令によって条件付きに変更されたすべての命令は条件付きで記述する必要があります。これらの条件は、IT 命令によって示された条件と一致する必要があります。例えば、ITTEE EQ 命令はその後に続く最初の 2 つの命令について EQ 条件を指定し、その次にある 2 つの命令には NE 条件を指定します。これら 4 つの命令は、それぞれ EQ、EQ、NE、NE 条件付きで記述する必要があります。

一部の命令は、IT 命令で条件付きにすることはできません。一部の命令は、IT ブロックの最後の命令である場合は条件付きにできますが、それ以外の場合は条件付きにできません。

条件フィールドを含む分岐命令は、IT 命令で条件付きにすることはできません。アセンブラ構文で、前の IT 命令と正しく一致する条件付き分岐が示されている場合、その命令は条件フィールドを含まない分岐命令エンコードを使用してアセンブルされます。

A4.2.2 UAL 命令構文でのラベルの使用

一部の命令の UAL 構文には、指定される命令から固定オフセットにある命令のラベル、またはリテラルデータ項目が含まれます。アセンブラは、次の動作を行う必要があります。

1. 命令の PC または Align(PC,4) の値を計算します。命令の PC の値は、Thumb 命令の場合は命令のアドレス + 4、ARM 命令の場合は + 8 です。命令の Align(PC,4) の値は、PC の値に 0xFFFFFFFC とのビット AND を適用し、ワードアラインドとした値です。ARM 命令については、PC と Align(PC,4) の値は同じですが、Thumb 命令の場合は異なる可能性があります。
2. 命令の PC または Align(PC,4) の値から、ラベルの付いている命令、またはリテラルデータ項目へのオフセットを計算します。
3. 命令の *PC 相対*エンコードをアセンブルします。つまり、PC または Align(PC,4) の値を読み出し、計算されたオフセットを加算すると、目的のアドレスが得られるようにします。

次の命令の構文にはラベルが含まれます。

- B、BL、BLX（イミディエート）。これらの命令のアセンブラ構文では、常に分岐先の命令のラベルが指定されます。これらの命令のエンコードでは、符号拡張されたイミディエートオフセットが指定され、そのオフセットが命令の PC の値に追加されて分岐先アドレスが計算されます。
- CBNZ および CBZ。これらの命令のアセンブラ構文では、常に分岐先の命令のラベルが指定されます。これらの命令のエンコードでは、ゼロ拡張されたイミディエートオフセットが指定され、そのオフセットが命令の PC の値に追加されて分岐先アドレスが計算されます。これらの命令は後方分岐をサポートしていません。
- LDC、LDC2、LDR、LDRB、LDRD、LDRH、LDRSB、LDRSH、PLD、PLI これらのロード命令の通常のアセンブラ構文では、ロードされるリテラルデータ項目のラベルを指定できます。これらの命令のエンコードでは、ゼロ拡張されたイミディエートオフセットが指定され、そのオフセットが命令の Align(PC,4) の値に加算または減算され、データ項目のアドレスが計算されます。このようなエンコードのいくつかは、オフセットの加算か減算のどちらかに固定されており、その操作が必要な場合のみ使用する必要がありますが、ほとんどのエンコードにはオフセットを加算するか減算するかを指定するビットが含まれています。

  アセンブラは、これらの命令について通常の構文のオフセットとして 0 を計算する場合、命令の Align(PC,4) の値に 0 を加算するエンコードをアセンブルする必要があります。Align(PC,4) の値から 0 を減算するエンコードは通常の構文では指定できません。

  これらの命令には、イミディエートオフセットを加算するか減算するかを明示的に指定する代替構文があります。この構文では、次に示すようにラベルを [PC, #+/-<imm>] に置き換えます。

  +/- Align(PC,4) の値にイミディエートオフセットを加算することを指定する場合は、+ を指定するか、符号を省略します。減算する場合は - を指定します。

  <imm> イミディエートオフセットを指定します。

  この代替構文では、Align(PC,4) の値から 0 を減算するエンコードをアセンブルすることが可能で、そのエンコードを逆アセンブルして、再度正しくアセンブル可能な構文に戻すこともできます。
- ADR。この命令の通常のアセンブラ構文では、命令またはリテラルデータ項目のラベルを指定し、アドレスを計算することができます。この命令のエンコードでは、ゼロ拡張されたイミディエートオフセットが指定され、そのオフセットが命令の Align(PC,4) の値に加算または減算され、データ項目のアドレスが計算されます。また、オペコードのいくつかのビットは、オフセットに対して加算と減算のどちらを行うかの指定に使用されます。

アセンブラは、これらの命令について通常の構文のオフセットとして 0 を使用して計算する場合、命令の Align(PC,4) の値に 0 を追加するエンコードをアセンブルする必要があります。Align(PC,4) の値から 0 を減算するエンコードは通常の構文では指定できません。

この命令には、イミディエート値を加算するか減算するかを明示的に指定する代替構文があります。この構文では、加算の場合 ADD <Rd>,PC,#<imm>、減算の場合 SUB <Rd>,PC,#<imm> と記述します。この代替構文では、Align(PC,4) の値から 0 を減算するエンコードをアセンブルすることが可能で、そのエンコードを逆アセンブルして、再度正しくアセンブル可能な構文に戻すこともできます。

注

次の構文の使用はできる限り避けることをお勧めします。

- ADR、LDC、LDC2、LDR、LDRB、LDRD、LDRH、LDRSB、LDRSH、PLD、PLI 命令の代替構文
- これらの命令について、Align(PC,4) の値から 0 を減算するエンコード

A4.3 分岐命令

Thumb 命令セットの分岐命令の要約を表 A4-1 に示します。実行のフローを変更する他、一部の分岐命令は命令セットを変更できます。

表 A4-1 分岐命令

命令	使用法	範囲
P.A6-40「*B*」	ターゲットアドレスへの分岐	+/–1 MB
P.A6-52「*CBNZ, CBZ*」	比較して非 0 で分岐、比較して 0 で分岐	0-126 B
P.A6-49「*BL*」	サブルーチン呼び出し	+/–16 MB
P.A6-50「*BLX（レジスタ）*」	サブルーチン呼び出し、オプションとして命令セットの変更	任意
P.A6-51「*BX*」	ターゲットアドレスへの分岐、命令セットの変更	任意
P.A6-257「*TBB*、*TBH*」	テーブル分岐（バイトオフセット）	0-510 B
	テーブル分岐（ハーフワードオフセット）	0-131070 B

LDR および LDM 命令で分岐を引き起こすこともできます。詳細については、P.A4-16「*ロード/ストア命令*」と P.A4-19「*複数ロード/ストア命令*」を参照して下さい。

A4.4 データ処理命令

主要なデータ処理命令は、次のいずれかのグループに属します。

- *標準データ処理命令*。このグループは、基本的なデータ処理操作を実行し、いくつかのバリアントで共通のフォーマットを持ちます。
- P.A4-10「*シフト命令*」
- P.A4-11「*乗算命令*」
- P.A4-12「*飽和命令*」
- P.A4-13「*パッキングおよびアンパッキング命令*」
- P.A4-14「*その他のデータ処理命令*」
- P.A4-14「*除算命令*」

A4.4.1 標準データ処理命令

これらの命令には一般に、デスティネーションレジスタ Rd、最初のオペランドレジスタ Rn、2 番目のオペランドがあります。2 番目のオペランドは、別のレジスタ Rm または修飾されたイミディエート定数です。

2 番目のオペランドが修飾されたイミディエート定数の場合、命令の 12 ビットを使ってエンコードされます。詳細については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

2 番目のオペランドが別のレジスタの場合、オプションとして次のいずれかのシフトを適用できます。

LSL	1 ～ 31 ビットの論理左シフト
LSR	1 ～ 32 ビットの論理右シフト
ASR	1 ～ 32 ビットの算術右シフト
ROR	1 ～ 31 ビットの右ローテート
RRX	拡張付き右ローテート（詳細については、P.A2-5「*シフトおよびローテート演算*」を参照して下さい）

Thumb コードでは、シフトのビット数は常に定数で、命令にエンコードされます。

結果をデスティネーションレジスタに置く他、これらの命令はオプションとして、演算の結果に従って条件コードフラグをセットできます。フラグをセットしない場合、前の命令でセットされたそれまでのフラグ状態が保持されます。

Thumb 命令セットの主要なデータ処理命令の要約を P.A4-9 表 A4-2 に示します。一般に、これらの各命令は第 A6 章 *「Thumb 命令の詳細」*で、2 つのセクションに分けて説明され、各セクションでは次の場合について説明されています。

- INSTRUCTION（イミディエート）: 2 番目のオペランドが修飾されたイミディエート定数の場合
- INSTRUCTION（レジスタ）: 2 番目のオペランドがレジスタ、または定数によりシフトされたレジスタの場合

表 A4-2 標準データ処理命令

ニーモニック	命令	注
ADC	キャリー付き加算	-
ADD	加算	Thumb-2 命令セットでは、修飾されたイミディエート定数、またはゼロ拡張された 12 ビットイミディエート定数が使用できます。
ADR	PC 相対アドレスの生成	最初のオペランドは PC で、2 番目のオペランドはイミディエート定数です。Thumb-2 命令セットでは、ゼロ拡張された 12 ビットイミディエート定数が使用されます。演算は加算または減算です。
AND	ビット単位の AND	-
BIC	ビット単位のビットクリア	-
CMN	比較否定	フラグをセットします。ADD と似ていますが、デスティネーションレジスタはありません。
CMP	比較	フラグをセットします。SUB と似ていますが、デスティネーションレジスタはありません。
EOR	ビット単位の排他的論理和	-
MOV	オペランドをデスティネーションへコピー	オペランドは 1 つだけで、これらの命令のほとんどには 2 番目のオペランドと同じオプションがあります。オペランドがシフトされたレジスタの場合、この命令は実質的に LSL、LSR、ASR、ROR 命令のいずれかになります。詳細については、P.A4-10「*シフト命令*」を参照して下さい。 Thumb-2 命令セットでは、修飾されたイミディエート定数、またはゼロ拡張された 16 ビットイミディエート定数が使用できます。
MVN	ビット単位の NOT	オペランドは 1 つだけで、これらの命令のほとんどには 2 番目のオペランドと同じオプションがあります。
ORN	ビット単位の OR NOT	-
ORR	ビット単位の OR	-
RSB	逆減算	2 番目のオペランドから最初のオペランドを減算します。これによって、定数やシフトされたレジスタからの減算を実行できます。
SBC	キャリー付き減算	-
SUB	減算	Thumb-2 命令セットでは、修飾されたイミディエート定数、またはゼロ拡張された 12 ビットイミディエート定数が使用できます。
TEQ	等価テスト	フラグをセットします。EOR と似ていますが、デスティネーションレジスタはありません。
TST	テスト	フラグをセットします。AND と似ていますが、デスティネーションレジスタはありません。

A4.4.2 シフト命令

Thumb 命令セットのシフト命令の要約を表 A4-3 に示します。

表 A4-3 シフト命令

命令	参照先
算術右シフト	P.A6-36「*ASR（イミディエート）*」
算術右シフト	P.A6-38「*ASR（レジスタ）*」
論理左シフト	P.A6-135「*LSL（イミディエート）*」
論理左シフト	P.A6-137「*LSL（レジスタ）*」
論理右シフト	P.A6-139「*LSR（イミディエート）*」
論理右シフト	P.A6-141「*LSR（レジスタ）*」
右ローテート	P.A6-193「*ROR（イミディエート）*」
右ローテート	P.A6-195「*ROR（レジスタ）*」
拡張付き右ローテート	P.A6-197「*RRX*」

A4.4.3 乗算命令

これらの命令は、符号付きまたは符号なしの数値に対して演算を行います。一部のタイプの演算では、オペランドが符号付き、符号なしのいずれでも結果は同じです。

- 符号付きと符号なしの数値で差のない乗算命令の要約を表 A4-4 に示します。

  結果の下位 32 ビットが使用され、上位ビットは破棄されます。
- 符号付き乗算命令の要約を表 A4-5 に示します。
- 符号なし乗算命令の要約を表 A4-6 に示します。

表 A4-4 汎用乗算命令

命令	動作（ビットの数）
P.A6-147「*MLA*」	32 = 32 + 32 x 32
P.A6-148「*MLS*」	32 = 32 – 32 x 32
P.A6-161「*MUL*」	32 = 32 x 32

表 A4-5 符号付き乗算命令

命令	動作（ビットの数）
P.A6-212「*SMLAL*」	64 = 64 + 32 x 32
P.A6-213「*SMULL*」	64 = 32 x 32

表 A4-6 符号なし乗算命令

命令	動作（ビットの数）
P.A6-267「*UMLAL*」	64 = 64 + 32 x 32
P.A6-268「*UMULL*」	64 = 32 x 32

A4.4.4 飽和命令

Thumb 命令セットの飽和命令の要約を表 A4-7 に示します。詳細については、P.A2-9「*飽和の擬似コードの詳細*」を参照して下さい。

表 A4-7 主要な飽和命令

命令	参照先	動作
符号付き飽和	P.A6-214「*SSAT*」	32 ビットの値を選択された範囲に飽和します。オプションとして、値をシフトすることもできます。
符号なし飽和	P.A6-269「*USAT*」	32 ビットの値を選択された範囲に飽和します。オプションとして、値をシフトすることもできます。

A4.4.5 パッキングおよびアンパッキング命令

Thumb 命令セットのパッキングおよびアンパッキング命令の要約を表 A4-8 に示します。

表 A4-8 パッキングおよびアンパッキング命令

命令	参照	動作
符号付きバイト拡張	P.A6-253「*SXTB*」	8 ビットを 32 ビットに拡張
符号付きハーフワード拡張	P.A6-255「*SXTH*」	16 ビットを 32 ビットに拡張
符号なしバイト拡張	P.A6-271「*UXTB*」	8 ビットを 32 ビットに拡張
符号なしハーフワード拡張	P.A6-273「*UXTH*」	16 ビットを 32 ビットに拡張

A4.4.6 その他のデータ処理命令

Thumb 命令セットに含まれるその他のデータ処理命令の要約を表 A4-9 に示します。これらの命令に含まれるイミディエート値は、単純なバイナリ数値です。

表 A4-9 その他のデータ処理命令

命令	参照先	注
ビットフィールドのクリア	P.A6-42「*BFC*」	-
ビットフィールドの挿入	P.A6-43「*BFI*」	-
先行ゼロカウント	P.A6-57「*CLZ*」	-
上位ハーフワード移動	P.A6-154「*MOVT*」	16 ビットのイミディエート値を上位ハーフワードへ移動します。下位ハーフワードは変更されません。
ビット反転	P.A6-189「*RBIT*」	-
ワードのバイト反転	P.A6-190「*REV*」	-
パックハーフワードのバイト反転	P.A6-191「*REV16*」	-
符号付きハーフワードのバイト反転	P.A6-192「*REVSH*」	-
符号付きビットフィールドの抽出	P.A6-207「*SBFX*」	-
符号なしビットフィールドの抽出	P.A6-265「*UBFX*」	-

A4.4.7 除算命令

ARMv7-M プロファイルでは、Thumb-2 命令セットに符号付きおよび符号なしの整数除算命令が含まれており、これらの命令はハードウェアで実装されます。命令の詳細については、次の項目を参照して下さい。

- P.A6-209「*SDIV*」
- P.A6-266「*UDIV*」

ARMv7-M プロファイルでは、CCR.DIV_0_TRP ビットによって、0 による除算のフォルトの検出が可能になります。

DZ == 0 0 による除算を行った場合、結果として 0 が返されます。

DZ == 1 SDIV および UDIV で 0 による除算を行った場合、未定義命令例外が生成されます。

CCR.DIV__0_TRP ビットはリセット時に 0 にクリアされます。

A4.5 ステータスレジスタ アクセス命令

MRS および MSR 命令は、*アプリケーションプログラム ステータスレジスタ* (APSR) の内容を汎用レジスタへ、またはその逆に移動します。

APSR については、P.A2-13「*アプリケーションプログラム ステータスレジスタ (APSR)*」を参照して下さい。

APSR の条件フラグは、一般にデータ処理命令の実行によりセットされ、条件付き命令の実行の制御に使用されます。ただし、MSR 命令を使用して明示的にフラグをセットでき、MRS 命令を使用して明示的に現在のフラグの状態を読み出すことができます。

ステータスレジスタ アクセス命令 CPS、MRS、MSR システムレベルで使用する方法の詳細については、第 B3 章 *「ARMv7-M システム命令」*を参照して下さい。

A4.6 ロード / ストア命令

Thumb 命令セットに含まれる汎用レジスタのロード / ストア命令の要約を表 A4-10 に示します。P.A4-19「*複数ロード / ストア命令*」も参照して下さい。

ロード / ストア命令には、メモリのアドレスを指定するいくつかのオプションがあります。詳細については、P.A4-18「*アドレシングモード*」を参照して下さい。

表 A4-10 ロード / ストア命令

データタイプ	ロード	ストア	非特権ロード	非特権ストア	排他ロード	排他ストア
32 ビットワード	LDR	STR	LDRT	STRT	LDREX	STREX
16 ビットハーフワード	-	STRH	-	STRHT	-	STREXH
16 ビット符号なしハーフワード	LDRH	-	LDRHT	-	LDREXH	-
16 ビット符号付きハーフワード	LDRSH	-	LDRSHT	-	-	-
8 ビットバイト	-	STRB	-	STRBT	-	STREXB
8 ビット符号なしバイト	LDRB	-	LDRBT	-	LDREXB	-
8 ビット符号付きバイト	LDRSB	-	LDRSBT	-	-	-
2 つの 32 ビットワード	LDRD	STRD	-	-	-	-

A4.6.1 PC へのロード

LDR 命令は、値を PC へロードするために使用できます。ロードされた値は、P.A2-11「*ARM コアレジスタ演算の擬似コードの詳細*」の LoadWritePC() 擬似コード関数で説明されているように、インターワーキングアドレスとみなされます。

A4.6.2 ハーフワードとバイトのロード / ストア

ハーフワードとバイトのストアは、レジスタの下位ハーフワードまたはバイトを、それぞれ 16 または 8 ビットのメモリへストアします。ストアには符号付きと符号なしの区別はありません。

ハーフワードとバイトのロードは、16 または 8 ビットのメモリから、レジスタの下位ハーフワードまたはバイトへロードします。符号なしロードの場合はロードされた値が 32 ビットにゼロ拡張され、符号付きロードの場合は値が 32 ビットに符号拡張されます。

A4.6.3 非特権ロード / ストア

非特権モードでは、非特権のロード / ストアは対応する通常の操作と全く同じに動作します。特権モードでは、非特権ロード / ストアは非特権モードで実行されたものとして扱われます。詳細については、P.A3-29「*データアクセスの特権レベルアクセス制御*」を参照して下さい。

A4.6.4 排他ロード / ストア

排他ロード / ストアは共有メモリの同期に使用できます。詳細については、P.A3-9「*同期化とセマフォ*」を参照して下さい。

A4.6.5 アドレシングモード

ロード/ストアのアドレスは、ベースレジスタの値とオフセットの 2 つの部分から生成されます。

ベースレジスタには、任意の汎用レジスタが使用できます。

ロードの場合、PC をベースレジスタとすることができます。これによって、位置独立コードのための PC 相対アドレシングを使用できます。第 A6 章 *「Thumb 命令の詳細」*でタイトルに（リテラル）とマークされている命令は PC 相対ロードです。

オフセットは、次の 3 つのうちいずれかのフォーマットです。

イミディエート オフセットは符号なし数値で、ベースレジスタの値に加算または減算できます。イミディエートオフセット アドレシングは、データオブジェクトの先頭から一定の距離にあるデータエレメント、例えば構造体のフィールド、スタックオフセット、入力 / 出力レジスタなどにアクセスするのに便利です。

レジスタ オフセットは汎用レジスタの値です。このレジスタに PC は使用できません。この値はベースレジスタの値に加算または減算できます。レジスタオフセットは、配列やデータブロックにアクセスするのに便利です。

スケーリングされたレジスタ

オフセットは PC 以外の汎用レジスタで、イミディエート値によってシフトされてから、ベースレジスタの値に加算または減算されます。つまり、配列のインデクスを、配列の各要素のサイズだけスケーリングできます。

オフセットとベースレジスタから、次に示す 3 つの異なるアドレシングモードでメモリアドレスを計算できます。

オフセット オフセットがベースレジスタに加算または減算され、メモリアドレスが計算されます。

プリインデクス オフセットがベースレジスタに加算または減算され、メモリアドレスが計算されます。その後で、ベースレジスタはその新しいアドレスに更新されるため、配列やメモリブロックを順番にアクセスする動作を自動的に実行できます。

ポストインデクス ベースレジスタの値が、そのままメモリアドレスとして使用されます。その後で、ベースレジスタにオフセットが加算または減算されます。この値はベースレジスタに書き戻されるため、配列やメモリブロックを順番にアクセスする動作を自動的に実行できます。

注

すべての命令についてすべてのバリアントが使用できるわけではありません。また、イミディエート値に使用できる値の範囲と、スケーリングされるレジスタのオプションは、命令によって異なります。各命令での詳細については、第 A6 章 *「Thumb 命令の詳細」*を参照して下さい。

A4.7 複数ロード / ストア命令

複数ロード命令は、汎用レジスタのサブセットまたはすべてに、メモリからロードします。

複数ストア命令は、汎用レジスタのサブセットまたはすべてを、メモリへストアします。

メモリ位置は、連続したワードアラインドのワードです。使用されるアドレスはベースレジスタから取得され、ベースレジスタの値の上方または下方とすることができます。オプションとして、転送されたデータの合計サイズでベースレジスタを更新することもできます。

ARM および Thumb-2 命令セットの複数ロード / ストア命令の要約を表 A4-11 に示します。

表 A4-11 複数ロード / ストア命令

命令	説明
複数ロード（ポストインクリメントすなわちフル下降）	P.A6-84「*LDM / LDMIA / LDMFD*」
複数ロード（プリデクリメントすなわち空上昇）	P.A6-87「*LDMDB / LDMEA*」
複数のレジスタをスタックからポップ[a]	P.A6-185「*POP*」
複数のレジスタをスタックへプッシュ[b]	P.A6-187「*PUSH*」
複数ストア（ポストインクリメントすなわち空上昇）	P.A6-217「*STM*、*STMIA*、*STMEA*」
複数ストア（プリデクリメントすなわちフル下降）	P.A6-219「*STMDB*、*STMFD*」

a. この命令は、SP をベースレジスタとして使用しベースレジスタを更新する場合の LDM 命令と等価です。
b. この命令は、SP をベースレジスタとして使用しベースレジスタを更新する場合の STMDB 命令と等価です。

A4.7.1 PC へのロード

LDM、LDMDB、POP 命令は、値を PC へロードするために使用できます。ロードされた値は、P.A2-11「*ARM コアレジスタ演算の擬似コードの詳細*」の LoadWritePC() 擬似コード関数で説明されているように、インターワーキングアドレスとみなされます。

A4.8 その他の命令

Thumb 命令セットに含まれるその他の命令の要約を表 A4-12 に示します。

表 A4-12 その他の命令

命令	参照先
排他クリア	P.A6-56「*CLREX*」
デバッグヒント	P.A6-67「*DBG*」
データメモリバリア	P.A6-68「*DMB*」
データ同期バリア	P.A6-70「*DSB*」
命令同期バリア	P.A6-76「*ISB*」
If Then（以後の命令を条件付きにする）	P.A6-78「*IT*」
操作なし	P.A6-168「*NOP*」
データのプリロード	P.A6-177「*PLD（イミディエート、リテラル）*」
	P.A6-179「*PLD（レジスタ）*」
命令のプリロード	P.A6-181「*PLI（イミディエート、リテラル）*」
	P.A6-183「*PLI（レジスタ）*」
イベント送信	P.A6-211「*SEV*」
スーパバイザコール	P.A6-251「*SVC（以前の SWI）*」
イベント待ち	P.A6-275「*WFE*」
割り込み待ち	P.A6-276「*WFI*」
イールド	P.A6-277「*YIELD*」

A4.9 例外生成命令

次の命令は、プロセッサ例外を生成することを特に目的としている命令です。

- スーパバイザコール（SVC、以前は SWI と呼ばれていました）命令は、SVC 例外を起こすために使用されます。これは、ユーザモードのコードが特権のオペレーティングシステムコードを呼び出すための主要な機構です。詳細については、P.B1-11「*例外モデル*」を参照して下さい。
- ブレークポイント (BKPT) 命令は、ソフトウェアブレークポイントを提供します。この命令は、デバッグの構成に従い、デバッグモニタ例外を生成するか、実行中のシステムを停止させます。詳細については、P.C1-12「*デバッグイベントの動作*」を参照して下さい。

A4.10 コプロセッサ命令

コプロセッサと通信するための命令には 3 つのタイプがあります。これらの命令により、プロセッサは次の動作を実行できます。

- コプロセッサのデータ処理動作を開始する。詳細については、P.A6-54「*CDP, CDP2*」を参照して下さい。
- 汎用レジスタとコプロセッサのレジスタとの間でデータを転送する。詳細については、次の項目を参照して下さい。
  - P.A6-143「*MCR*、*MCR2*」
  - P.A6-145「*MCRR, MCRR2*」
  - P.A6-155「*MRC, MRC2*」
  - P.A6-157「*MRRC*、*MRRC2*」
- コプロセッサのロード / ストア命令で使用されるアドレスを生成する。詳細については、次の項目を参照して下さい。
  - P.A6-80「*LDC*、*LDC2（イミディエート）*」
  - P.A6-82「*LDC*、*LDC2（リテラル）*」
  - P.A6-215「*STC*、*STC2*」

命令セットでは、各コプロセッサ命令に含まれている 4 ビットのフィールドで最大 16 のコプロセッサを区別でき、各コプロセッサに特定の番号が割り当てられます。

注

大規模なコプロセッサ命令セットが必要な場合、1 つのコプロセッサで 16 個の番号のうち 2 つ以上を使用できます。

コプロセッサは、コアプロセッサと同じ命令ストリームを実行し、コプロセッサ命令でない命令や、他のコプロセッサ用の命令は無視します。どのコプロセッサハードウェアでも実行できないコプロセッサ命令は、UsageFault 例外を生成し、その理由が次のように記録されます。

- コプロセッサアクセス　レジスタによってコプロセッサへのアクセスが拒否されている場合、UFSR.NOCP フラグがセットされ、コプロセッサが存在しないことを示します。
- コプロセッサへのアクセスが許可されているが、命令が未知である場合、UNDEFINED フラグがセットされ、命令が未定義であることを示します。

第 A5 章
Thumb® 命令セットのエンコード

本章では、Thumb-2 命令セットを紹介し、この命令セットで ARM プログラマモデルがどのように使用されるかについて説明します。本章は以下のセクションから構成されています。

A5.1 Thumb-2 命令セットのエンコード

Thumb 命令ストリームは、ハーフワードでアラインされたハーフワードのシーケンスです。そのストリーム内の各 Thumb 命令は、単一の 16 ビット命令か、2 つの連続するハーフワードで構成された 32 ビット命令のどちらかです。

デコードされるハーフワードのビット [15:11] が、次のいずれかの値であれば、そのハーフワードは 32 ビット命令の最初のハーフワードです。

- 0b11101
- 0b11110
- 0b11111

それ以外の場合、そのハーフワードは 16 ビット命令です。

16 ビット Thumb 命令のエンコードの詳細については、P.A5-5「*16 ビット Thumb 命令のエンコード*」を参照して下さい。

32 ビット Thumb 命令のエンコードの詳細については、P.A5-13「*32 ビット Thumb 命令のエンコード*」を参照して下さい。

A5.1.1 未定義および予測不能の命令セット空間

未割り当て命令の実行を試みると、次のいずれかの結果が発生します。

- 予測不能な動作。このような命令は、予測不能と記述されています。
- 未定義命令例外。このような命令は、未定義と記述されています。

命令の説明または本章の中で、未定義と記載されている命令は未定義です。

次のいずれかに該当する命令は予測不能です。

- 命令のエンコード図で (0) または (1) と表記されたビットがそれぞれ 0 または 1 ではなく、そのエンコードの擬似コードで、別の特別な条件が適用されることが示されていない。
- 命令の説明または本章で、予測不能と記載されている。

特に記載がない限り、次の規則が適用されます。

- あるアーキテクチャバリアントで導入された Thumb 命令は、それ以前のアーキテクチャバリアントでは予測不能または未定義です。
- 1 つ以上のアーキテクチャ拡張機能で提供されている Thumb 命令は、その拡張機能を含まない実装では予測不能または未定義です。

どちらの場合も、このような命令は ARMv6T2 以前のアーキテクチャバリアントの 32 ビット命令の場合は予測不能で、それ以外の場合は未定義です。

A5.1.2 レジスタ指定子として 0b1111 を使用する場合

通常、Thumb 命令ではレジスタ指定子として 0b1111 を使用することはできません。値として 0b1111 が許可される場合、各種の意味に使用される可能性があります。レジスタ読み出しの場合は、次のような意味になります。

- PC の値、つまり現在の命令 + 4 のアドレスを読み出す。テーブル分岐命令 TBB および TBH では、ベースレジスタを PC にすることができます。これによって、メモリ内で分岐テーブルを命令の直後に配置することができます (条件付き分岐命令 B<cond> などの一部の命令は、レジスタ指定子を使用せずに暗黙的に PC の値を読み出します)。

  注

  STC 命令でベースレジスタとして PC を使用することは、ARMv7 では推奨されません。

- ワードアラインドの PC の値を読み出す。これは、現在の命令 + 4 のアドレスを読み出し、その値のビット [1:0] を強制的に 0 に設定することを意味します。LDC、LDR、LDRB、LDRD プリインデクス、ライトバックなし)、LDRH、LDRSB、LDRSH の各命令では、ワードアラインドの PC をベースレジスタにすることができます。これによって、PC 相対のデータアドレシングが可能になります。さらに、ADD 命令および SUB 命令の一部のエンコードでは、同じ目的でソースレジスタを 0b1111 にすることができます。

- 0 を読み出す。この動作は、その命令が、別のより汎用的な命令の特殊なケースで、1 つのオペランドが 0 である一部の場合に発生します。この場合、その命令の説明はより汎用的な命令用の擬似コードにおけるひとつの特殊なケースとして別のページに記載され、相互参照のページが示されています。

レジスタ書き込みの場合は、次のような意味になります。

- LDR 命令のデスティネーションレジスタとして PC を指定できます。この場合、Rt が 0b1111 としてエンコードされます。ロードした値はアドレスとして扱われ、実行の結果そのアドレスへの分岐が行われます。ロードされた値のビット [0] は分岐後の実行状態の選択に使用され、この値は 1 の必要があります。

  他の一部の命令では、暗黙的に (B<cond> など)、またはレジスタ指定子の代わりにレジスタマスクを使用して (LDM)、同様に PC に書き込むことができます。分岐先のアドレスは、ロードした値 (LDM など)、レジスタ値 (BX など)、または計算の結果 (TBB や TBH など) にすることができます。

- 計算の結果を破棄する。この動作は、その命令が、別のより汎用的な命令の特殊なケースで、結果が破棄される場合に発生します。この場合、その命令の説明は別のページに記載され、より汎用的な命令の説明において、擬似コードの特殊なケースとしてそのページへの参照が示されています。

- LDRB、LDRH、LDRSB、LDRSH 命令のデスティネーションレジスタ指定子が 0b1111 の場合、その命令はロード操作ではなくメモリヒントです。

- MRC 命令のデスティネーションレジスタ指定子が 0b1111 の場合は、コプロセッサから転送された値のビット [31:28] が APSR の N、Z、C、V フラグに書き込まれ、ビット [27:0] は破棄されます。

A5.1.3 レジスタ指定子として 0b1101 を使用する場合

R13 は、主にスタックポインタとして使用するように Thumb 命令セット内で定義されており、Thumb 命令では一般に SP と表記されます。32 ビット Thumb 命令で、このセクションに記載されているアーキテクチャ定義の制限を無視して R13 を汎用レジスタとして使用した場合、結果は予測不能です。

R13 に適用される制限については、以下を参照して下さい。

- P.A5-4「*R13[1:0] の定義*」
- P.A5-4「*32 ビット Thumb 命令での R13 のサポート*」

*16 ビット Thumb 命令での R13 のサポート*も参照して下さい。

R13[1:0] の定義

R13 のビット [1:0] は、*SBZP*（常に 0 または保持）として扱われます。ビット [1:0] に 0 以外の値を書き込んだ場合、動作は予測不能です。ビット [1:0] を読み出した場合、0 が返されます。

32 ビット Thumb 命令での R13 のサポート

R13 命令のサポートは、次の操作に制限されます。

- MOV 命令のソースレジスタまたはデスティネーションレジスタとして R13 を使用する。シフトを伴わないレジスタ間転送のみサポートされ、フラグはセットされません。

```
MOV     SP,Rm
MOV     Rn,SP
```

- アライメントの倍数だけ R13 を加算または減算して調整する。

```
ADD{W}  SP,SP,#N            ; For N a multiple of 4
SUB{W}  SP,SP,#N            ; For N a multiple of 4
ADD     SP,SP,Rm,LSL #shft  ; For shft=0,1,2,3
SUB     SP,SP,Rm,LSL #shft  ; For shft=0,1,2,3
```

- 任意のロード / ストア命令のベースレジスタ (Rn) として R13 を使用する。これによって、ロード命令、ストア命令、メモリヒント命令で、SP をベースとしたアドレシング（正または負のオフセット、ライトバックありまたはなし）がサポートされます。

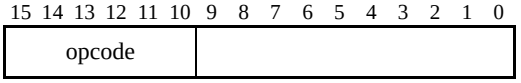
- 任意の ADD{S}、CMN、CMP、SUB{S} 命令で、最初のオペランド (Rn) として R13 を使用する。加算命令と減算命令では、SP をベースとしたアドレスを生成し、そのアドレスを汎用レジスタへ転送する操作がサポートされます。CMN および CMP は、特定の状況でスタックのチェックを行うのに便利です。

- 任意の LDR 命令または STR 命令で、転送先レジスタ (Rt) として R13 を使用する。

16 ビット Thumb 命令での R13 のサポート

上位レジスタを使用する 16 ビットデータ処理命令の場合、*32 ビット Thumb 命令での R13 のサポート*で説明されている方法でのみ R13 を使用できます。それ以外の使用は推奨されません。これは CMP と ADD の上位レジスタ形式に影響し、R13 を Rm として使用することは推奨されません。

A5.2 16 ビット Thumb 命令のエンコード

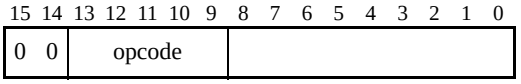


16 ビット命令のエンコードの割り当てを、表 A5-1 に示します。

表 A5-1 16 ビット Thumb 命令のエンコード

opcode	命令または命令クラス
00xxxx	P.A5-6「*シフト（イミディエート）、加算、減算、移動、比較*」
010000	P.A5-7「*データ処理*」
010001	P.A5-8「*特殊なデータ命令および分岐交換*」
01001x	リテラルプールからのロード。P.A6-91「*LDR（リテラル）*」参照
0101xx	P.A5-9「*単一のデータ項目のロード/ ストア*」
011xxx	
100xxx	
10100x	PC 相対アドレスの生成。P.A6-30「*ADR*」参照
10101x	SP 相対アドレスの生成。P.A6-26「*ADD（SP + イミディエート）*」参照
1011xx	P.A5-10「*その他の 16 ビット命令*」
11000x	複数のレジスタのストア。P.A6-217「*STM*、*STMIA*、*STMEA*」参照
11001x	複数のレジスタのロード。P.A6-84「*LDM / LDMIA / LDMFD*」参照
1101xx	P.A5-12「*条件付き分岐およびスーパバイザコール*」
11100x	無条件分岐。P.A6-40「*B*」参照

A5.2.1 シフト（イミディエート）、加算、減算、移動、比較

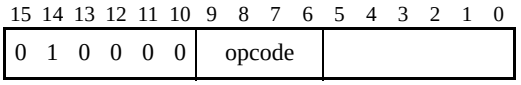


この空間でのエンコードの割り当てを、表 A5-2 に示します。

表 A5-2 16 ビット Thumb-2 命令のエンコード

opcode	命令	参照先
000xx	論理左シフト	P.A6-135「*LSL（イミディエート）*」
001xx	論理右シフト	P.A6-139「*LSR（イミディエート）*」
010xx	算術右シフト	P.A6-36「*ASR（イミディエート）*」
01100	レジスタの加算	P.A6-24「*ADD（レジスタ）*」
01101	レジスタの減算	P.A6-245「*SUB（レジスタ）*」
01110	3 ビットイミディエート値の加算	P.A6-22「*ADD（イミディエート）*」
01111	3 ビットイミディエート値の減算	P.A6-243「*SUB（イミディエート）*」
100xx	移動	P.A6-149「*MOV（イミディエート）*」
101xx	比較	P.A6-62「*CMP（イミディエート）*」
110xx	8 ビットイミディエート値の加算	P.A6-22「*ADD（イミディエート）*」
111xx	8 ビットイミディエート値の減算	P.A6-243「*SUB（イミディエート）*」

A5.2.2 データ処理

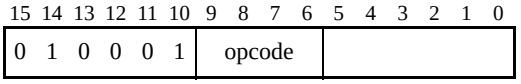


この空間でのエンコードの割り当てを、表 A5-3 に示します。

表 A5-3 16 ビット Thumb-2 データ処理命令

opcode	命令	参照先
0000	ビット単位 AND	P.A6-34「*AND（レジスタ）*」
0001	排他的論理和	P.A6-74「*EOR（レジスタ）*」
0010	論理左シフト	P.A6-137「*LSL（レジスタ）*」
0011	論理右シフト	P.A6-141「*LSR（レジスタ）*」
0100	算術右シフト	P.A6-38「*ASR（レジスタ）*」
0101	キャリー付き加算	P.A6-20「*ADC（レジスタ）*」
0110	キャリー付き減算	P.A6-205「*SBC（レジスタ）*」
0111	右ローテート	P.A6-195「*ROR（レジスタ）*」
1000	ビット単位の AND でフラグをセット	P.A6-263「*TST（レジスタ）*」
1001	0 からの逆減算	P.A6-199「*RSB（イミディエート）*」
1010	レジスタの比較	P.A6-64「*CMP（レジスタ）*」
1011	比較否定	P.A6-60「*CMN（レジスタ）*」
1100	論理和	P.A6-175「*ORR（レジスタ）*」
1101	2 つのレジスタの乗算	P.A6-161「*MUL*」
1110	ビットクリア	P.A6-46「*BIC（レジスタ）*」
1111	ビット単位 NOT	P.A6-165「*MVN（レジスタ）*」

A5.2.3 特殊なデータ命令および分岐交換

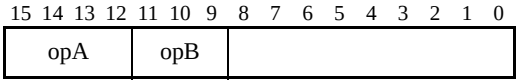


この空間でのエンコードの割り当てを、表 A5-4 に示します。

表 A5-4 特殊なデータ命令および分岐交換

opcode	命令	参照先
00xx	レジスタの加算	P.A6-24「*ADD（レジスタ）*」
0100	予測不能	
0101	レジスタの比較	P.A6-64「*CMP（レジスタ）*」
011x		
10xx	レジスタの移動	P.A6-151「*MOV（レジスタ）*」
110x	分岐交換	P.A6-51「*BX*」
111x	リンク付き分岐交換	P.A6-50「*BLX（レジスタ）*」

A5.2.4 単一のデータ項目のロード / ストア



これらの命令の opA は、次のいずれかの値です。

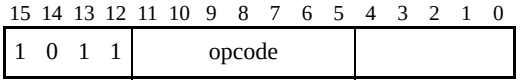
- 0b0101
- 0b011x
- 0b100x

この空間でのエンコードの割り当てを、表 A5-5 に示します。

表 A5-5 16 ビット Thumb-2 ロード / ストア命令

opA	opB	命令	参照先
0101	000	レジスタストア	P.A6-223「*STR（レジスタ）*」
0101	001	レジスタストア ハーフワード	P.A6-239「*STRH（レジスタ）*」
0101	010	レジスタストア バイト	P.A6-227「*STRB（レジスタ）*」
0101	011	レジスタロード符号付きバイト	P.A6-123「*LDRSB（レジスタ）*」
0101	100	レジスタロード	P.A6-93「*LDR（レジスタ）*」
0101	101	レジスタロード ハーフワード	P.A6-115「*LDRH（レジスタ）*」
0101	110	レジスタロード バイト	P.A6-99「*LDRB（レジスタ）*」
0101	111	レジスタロード符号付きハーフワード	P.A6-131「*LDRSH（レジスタ）*」
0110	0xx	レジスタストア	P.A6-221「*STR（イミディエート）*」
0110	1xx	レジスタロード	P.A6-89「*LDR（イミディエート）*」
0111	0xx	レジスタストア バイト	P.A6-225「*STRB（イミディエート）*」
0111	1xx	レジスタストア バイト	P.A6-95「*LDRB（イミディエート）*」
1000	0xx	レジスタストア ハーフワード	P.A6-237「*STRH（イミディエート）*」
1000	1xx	レジスタロード ハーフワード	P.A6-111「*LDRH（イミディエート）*」
1001	0xx	レジスタストア SP 相対	P.A6-221「*STR（イミディエート）*」
1001	1xx	レジスタロード SP 相対	P.A6-89「*LDR（イミディエート）*」

A5.2.5 その他の 16 ビット命令

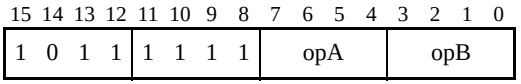


この空間でのエンコードの割り当てを、表 A5-6 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-6 その他の 16 ビット命令

opcode	命令	参照先
0110011	プロセッサ状態変更	P.B3-2「*CPS*」
00000xx	SP へのイミディエート値の加算	P.A6-26「*ADD (SP + イミディエート)*」
00001xx	SP からのイミディエート値の減算	P.A6-247「*SUB (SP - イミディエート)*」
0001xxx	比較して 0 で分岐	P.A6-52「*CBNZ, CBZ*」
001000x	符号付きハーフワード拡張	P.A6-255「*SXTH*」
001001x	符号付きバイト拡張	P.A6-253「*SXTB*」
001010x	符号なしハーフワード拡張	P.A6-273「*UXTH*」
001011x	符号なしバイト拡張	P.A6-271「*UXTB*」
0011xxx	比較して 0 で分岐	P.A6-52「*CBNZ, CBZ*」
010xxxx	複数レジスタプッシュ	P.A6-187「*PUSH*」
1001xxx	比較して非 0 で分岐	P.A6-52「*CBNZ, CBZ*」
101000x	ワードのバイト反転	P.A6-190「*REV*」
101001x	パックされたハーフワードのバイト反転	P.A6-191「*REV16*」
101011x	符号付きハーフワードのバイト反転	P.A6-192「*REVSH*」
1011xxx	比較して非 0 で分岐	P.A6-52「*CBNZ, CBZ*」
110xxxx	複数レジスタポップ	P.A6-185「*POP*」
1110xxx	ブレークポイント	P.A6-48「*BKPT*」
1111xxx	If-Then とヒント	P.A5-11「*If-Then とヒント*」

If-Then とヒント



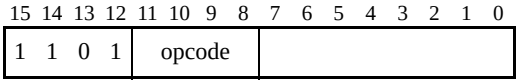
この空間でのエンコードの割り当てを、表 A5-7 に示します。

この空間に含まれる他のエンコードは未割り当てのヒントです。NOP として実行されますが、ソフトウェアでは使用しないで下さい。

表 A5-7 その他の 16 ビット命令

opA	opB	命令	参照先
xxxx	0000 以外	If-Then	P.A6-78「*IT*」
0000	0000	無操作ヒント	P.A6-168「*NOP*」
0001	0000	放棄ヒント	P.A6-277「*YIELD*」
0010	0000	イベント待ちヒント	P.A6-275「*WFE*」
0011	0000	割り込み待ちヒント	P.A6-276「*WFI*」
0100	0000	イベント送信ヒント	P.A6-211「*SEV*」

A5.2.6 条件付き分岐およびスーパバイザコール

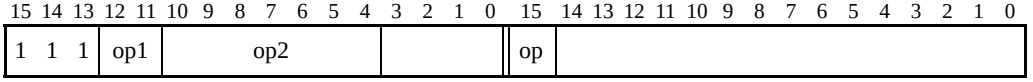


この空間でのエンコードの割り当てを、表 A5-8 に示します。

表 A5-8 分岐命令およびスーパバイザコール命令

opcode	命令	参照先
111x 以外	条件付き分岐	P.A6-40「*B*」
1110	恒久的に未定義	
1111	スーパバイザコール	P.A6-251「*SVC（以前の SWI）*」

A5.3 32 ビット Thumb 命令のエンコード



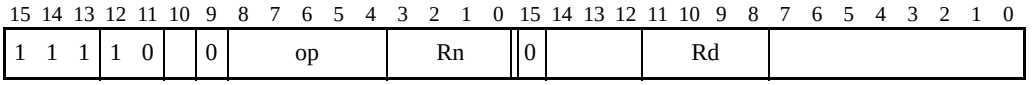
op1 != 0b00 です。op1 == 0b00 の場合は、16 ビット命令がエンコードされます。詳細については、P.A5-5「*16 ビット Thumb 命令のエンコード*」を参照して下さい。

この空間での ARMv7-M Thumb エンコードの割り当てを、表 A5-9 に示します。

表 A5-9 32 ビット Thumb-2 命令のエンコード

op1	op2	op	命令クラス
01	00xx 0xx	x	P.A5-20「*複数ロード/ ストア*」
01	00xx 1xx	x	P.A5-21「*ダブルワードのロード/ ストア、排他ロード/ ストア、テーブル分岐*」
01	01xx xxx	x	P.A5-26「*データ処理（シフトしたレジスタ）*」
01	1xxx xxx	x	P.A5-31「*コプロセッサ命令*」
10	x0xx xxx	0	P.A5-14「*データ処理（修飾イミディエート）*」
10	x1xx xxx	0	P.A5-17「*データ処理（普通のバイナリ イミディエート）*」
10	xxxx xxx	1	P.A5-18「*分岐およびその他の制御*」
11	000x xx0	x	P.A5-25「*単一データ項目のストア*」
11	00xx 001	x	P.A5-24「*バイトのロード、メモリヒント*」
11	00xx 011	x	P.A5-23「*ハーフワードのロード、未割り当てメモリヒント*」
11	00xx 101	x	P.A5-22「*ワードのロード*」
11	00xx 111	x	未定義
11	010x xxx	x	P.A5-27「*データ処理（レジスタ）*」
11	0110 xxx	x	P.A5-29「*乗算、積和演算、および絶対値の差*」
11	0111 xxx	x	P.A5-30「*ロング乗算、ロング積和演算、および除算*」
11	1xxx xxx	x	P.A5-31「*コプロセッサ命令*」

A5.3.1 データ処理（修飾イミディエート）



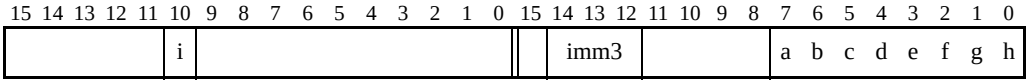
この空間でのエンコードの割り当てを、表 A5-10 に示しています。この空間に含まれる他のエンコードは未定義 です。

表 A5-10 32 ビット修飾イミディエート値データ処理命令

op	Rn	Rd	命令	参照先
0000x		1111 以外	ビット単位論理積	P.A6-32「*AND（イミディエート）*」
		1111	テスト	P.A6-261「*TST（イミディエート）*」
0001x			ビット単位クリア	P.A6-44「*BIC（イミディエート）*」
0010x	1111 以外		ビット単位論理和	P.A6-173「*ORR（イミディエート）*」
	1111		移動	P.A6-149「*MOV（イミディエート）*」
0011x	1111 以外		ビット単位否定論理和	P.A6-169「*ORN（イミディエート）*」
	1111		ビット単位否定	P.A6-163「*MVN（イミディエート）*」
0100x		1111 以外	ビット単位排他的論理和	P.A6-72「*EOR（イミディエート）*」
		1111	等価テスト	P.A6-259「*TEQ（イミディエート）*」
1000x		1111 以外	加算	P.A6-22「*ADD（イミディエート）*」
		1111	比較否定	P.A6-58「*CMN（イミディエート）*」
1010x			キャリー付き加算	P.A6-18「*ADC（イミディエート）*」
1011x			キャリー付き減算	P.A6-203「*SBC（イミディエート）*」
1101x		1111 以外	減算	P.A6-243「*SUB（イミディエート）*」
		1111	比較	P.A6-62「*CMP（イミディエート）*」
1110x			逆減算	P.A6-199「*RSB（イミディエート）*」

これらの命令はすべて、単純な 12 ビットバイナリ数値ではなく、修飾イミディエート定数を使用します。これによって、値の範囲がより広く応用可能です。詳細については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

A5.3.2 Thumb-2 命令での修飾イミディエート定数



Thumb-2 データ処理命令で使用可能な修飾イミディエート定数の範囲と、それらがこの命令内の a、b、c、d、e、f、g、h、i、imm3 の各フィールドにどのようにエンコードされるかを、表 A5-11 に示します。

表 A5-11 Thumb-2 データ処理命令での修飾イミディエート値のエンコード

< 定数 > [a]	i	imm3	a
00000000 00000000 00000000 abcdefgh	0	000	a
00000000 abcdefgh 00000000 abcdefgh [b]	0	001	a
abcdefgh 00000000 abcdefgh 00000000 [b]	0	010	a
abcdefgh abcdefgh abcdefgh abcdefgh [b]	0	011	a
1bcdefgh 00000000 00000000 00000000	0	100	0
01bcdefg h0000000 00000000 00000000	0	100	1
001bcdef gh000000 00000000 00000000	0	101	0
0001bcde fgh00000 00000000 00000000	0	101	1
... （他の位置にシフトされた 8 ビット値）	...	...	...
00000000 00000000 000001bc defgh000	1	110	1
00000000 00000000 0000001b cdefgh00	1	111	0
00000000 00000000 00000001 bcdefgh0	1	111	1

a. この表では、abcdefgh をエンコード図に関連付けるため、イミディエート値をバイナリ形式で示しています。アセンブリ構文では、イミディエート値を通常の方法（デフォルトの 10 進数）で指定します。
b. abcdefgh == 00000000 の場合、結果は予測不能です。

キャリーアウト

i == 0 および imm3 == 0b000 での論理演算は、キャリーフラグに影響を与えません。それ以外の場合、フラグをセットする論理演算によって、キャリーフラグが修飾イミディエート定数のビット [31] の値に設定されます。

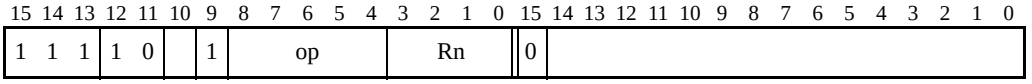
動作

```
// ThumbExpandImm()
// ================

bits(32) ThumbExpandImm(bits(12) imm12)

    // APSR.C argument to following function call does not affect the imm32 result.
    (imm32, -) = ThumbExpandImm_C(imm12, APSR.C);

    return imm32;
```



```
// ThumbExpandImm_C()
// ==================

(bits(32), bit) ThumbExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then

        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;

    else

        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

A5.3.3 データ処理（普通のバイナリ イミディエート）



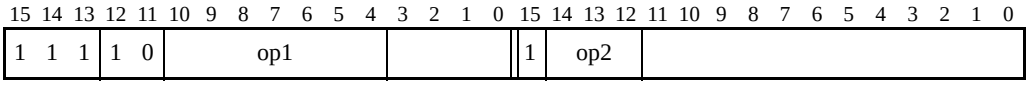
この空間でのエンコードの割り当てを、P.A5-14 表 A5-10 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-12 32 ビット未修飾イミディエート値データ処理命令

op	Rn	命令	参照先
00000	1111 以外	ワイド値（12 ビット）の加算	P.A6-22「*ADD（イミディエート）*」
	1111	PC 相対アドレスの生成	P.A6-30「*ADR*」
00100		ワイド値（16 ビット）の移動	P.A6-149「*MOV（イミディエート）*」
01010	1111 以外	ワイド値（12 ビット）の減算	P.A6-243「*SUB（イミディエート）*」
	1111	PC 相対アドレスの生成	P.A6-30「*ADR*」
01100		上位ビット（16 ビット）の移動	P.A6-154「*MOVT*」
100x0 [a]		符号付き飽和	P.A6-214「*SSAT*」
10100		符号付きビットフィールドの抽出	P.A6-207「*SBFX*」
10110	1111 以外	ビットフィールドの挿入	P.A6-43「*BFI*」
	1111	ビッドフィールドのクリア	P.A6-42「*BFC*」
110x0 [a]		符号なし飽和	P.A6-269「*USAT*」
11100		符号なしビットフィールドの抽出	P.A6-265「*UBFX*」

a. 命令の 2 番目のハーフワードでは、ビット [14:12.7:6] != 0b00000 です。

A5.3.4 分岐およびその他の制御

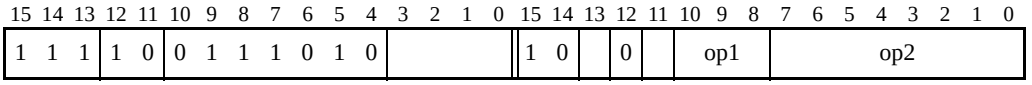


この空間でのエンコードの割り当てを、表 A5-13 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-13 分岐命令およびその他の制御命令

op2	op1	命令	参照先
0x0	x111xxx 以外	条件付き分岐	P.A6-40「*B*」
0x0	011100x	特殊レジスタへの移動	P.A6-160「*MSR（レジスタ）*」
0x0	0111010	-	P.A5-19「*ヒント命令*」
0x0	0111011	-	P.A5-19「*その他の制御命令*」
0x0	011111x	特殊レジスタからの移動	P.A6-159「*MRS*」
010	1111111	恒久的に未定義	-
0x1	xxxxxxx	分岐	P.A6-40「*B*」
1x0	xxxxxxx		
1x1	xxxxxxx	リンク付き分岐	P.A6-49「*BL*」

ヒント命令



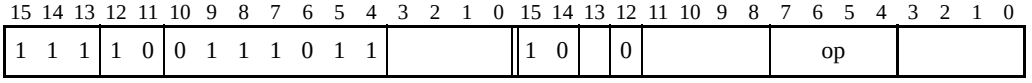
この空間でのエンコードの割り当てを、表 A5-14 に示します。この空間に含まれる他のエンコードは未割り当てヒントで、NOP として実行されます。これらの未割り当てヒントのエンコードは予約されているため、ソフトウェアで使用しないで下さい。

表 A5-14 プロセッサ状態変更命令およびヒント命令

op1	op2	命令	参照先
000 以外	xxxx xxxx	未定義[a]	
000	0000 0000	無操作ヒント	P.A6-168「*NOP*」
000	0000 0001	イールドヒント	P.A6-277「*YIELD*」
000	0000 0010	イベント待ちヒント	P.A6-275「*WFE*」
000	0000 0011	割り込み待ちヒント	P.A6-276「*WFI*」
000	0000 0100	イベント送信ヒント	P.A6-211「*SEV*」
000	1111 xxxx	デバッグヒント	P.A6-67「*DBG*」

a. これらのエンコードによって、ARMv7-A および ARMv7-R のアーキテクチャプロファイルでの 32 ビット形式の CPS 命令が提供されます。

その他の制御命令

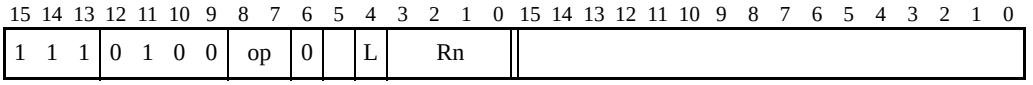


この空間でのエンコードの割り当てを、表 A5-15 に示します。この空間に含まれる他のエンコードは、ARMv7-M では未定義です。

表 A5-15 その他の制御命令

op	命令	参照先
0010	排他クリア	P.A6-56「*CLREX*」
0100	データ同期バリア	P.A6-70「*DSB*」
0101	データ メモリバリア	P.A6-68「*DMB*」
0110	命令同期バリア	P.A6-76「*ISB*」

A5.3.5 複数ロード / ストア

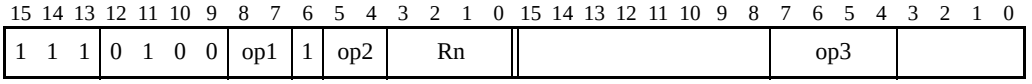


この空間でのエンコードの割り当てを、表 A5-16 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-16 複数ロード / ストア命令

op	L	Rn	命令	参照先
01	0		複数ストア（ポストインクリメント、空上昇）	P.A6-217「*STM*、*STMIA*、*STMEA*」
01	1	1101 以外	複数ロード（ポストインクリメント、フル下降）	P.A6-84「*LDM / LDMIA / LDMFD*」
01	1	1101	スタックからの複数レジスタのポップ	P.A6-185「*POP*」
10	0	1101 以外	複数ストア（プリデクリメント、フル下降）	P.A6-219「*STMDB*、*STMFD*」
10	0	1101	スタックへの複数レジスタのプッシュ	P.A6-187「*PUSH*」
10	1		複数ロード（プリデクリメント、空上昇）	P.A6-87「*LDMDB / LDMEA*」

A5.3.6 ダブルワードのロード / ストア、排他ロード / ストア、テーブル分岐

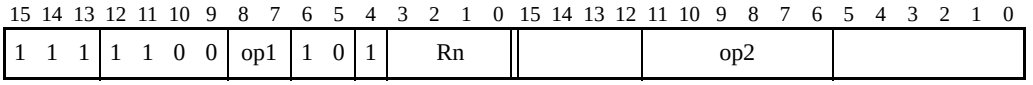


この空間でのエンコードの割り当てを、表 A5-17 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-17 ダブルまたは排他ロード / ストア、テーブル分岐

op1	op2	op3	命令	参照先
00	00	xxxx	排他レジスタストア	P.A6-233「*STREX*」
00	01	xxxx	排他レジスタロード	P.A6-107「*LDREX*」
0x 1x	10 x0	xxxx xxxx	ダブルレジスタ ストア	P.A6-231「*STRD（イミディエート）*」
0x 1x	11 x1	xxxx xxxx	ダブルレジスタ ロード	P.A6-103「*LDRD（イミディエート）*」, P.A6-105「*LDRD（リテラル）*」
01	00	0100	排他レジスタストア バイト	P.A6-234「*STREXB*」
01	00	0101	排他レジスタストア ハーフワード	P.A6-235「*STREXH*」
01	01	0000	テーブル分岐バイト	P.A6-257「*TBB*、*TBH*」
01	01	0001	テーブル分岐ハーフワード	P.A6-257「*TBB*、*TBH*」
01	01	0100	排他レジスタロード バイト	P.A6-108「*LDREXB*」
01	01	0101	排他レジスタロード ハーフワード	P.A6-109「*LDREXH*」

A5.3.7 ワードのロード

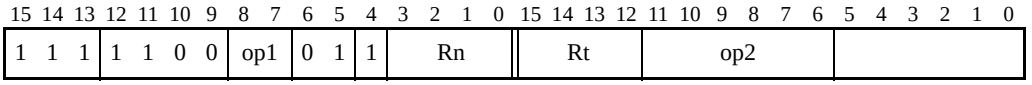


この空間でのエンコードの割り当てを、表 A5-18 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-18 ワードのロード

op1	op2	Rn	命令	参照
01	xxxxxx	1111 以外	レジスタロード	P.A6-89「*LDR（イミディエート）*」
00	1xx1xx	1111 以外		
00	1100xx	1111 以外		
00	1110xx	1111 以外	非特権レジスタロード	P.A6-134「*LDRT*」
00	000000	1111 以外	レジスタロード	P.A6-93「*LDR（レジスタ）*」
0x	xxxxxx	1111	レジスタロード	P.A6-91「*LDR（リテラル）*」

A5.3.8 ハーフワードのロード、未割り当てメモリヒント



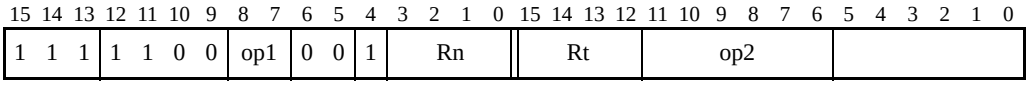
この空間でのエンコードの割り当てを、表 A5-19 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-19 ハーフワードのロード

op1	op2	Rn	Rt	命令	参照先
01	xxxxxx	1111 以外	1111 以外	レジスタロード ハーフワード	P.A6-111「*LDRH（イミディエート）*」
00	1xx1xx	1111 以外	1111 以外		
00	1100xx	1111 以外	1111 以外		
00	1110xx	1111 以外	1111 以外	非特権レジスタロード ハーフワード	P.A6-117「*LDRHT*」
0x	xxxxxx	1111	1111 以外	レジスタロード符号付きハーフワード	P.A6-113「*LDRH（リテラル）*」
00	000000	1111 以外	1111 以外	レジスタロード符号付きハーフワード	P.A6-115「*LDRH（レジスタ）*」
11	xxxxxx	1111 以外	1111 以外	レジスタロード符号付きハーフワード	P.A6-127「*LDRSH（イミディエート）*」
10	1xx1xx	1111 以外	1111 以外		
10	1100xx	1111 以外	1111 以外		
10	1110xx	1111 以外	1111 以外	非特権レジスタロード符号付きハーフワード	P.A6-133「*LDRSHT*」
1x	xxxxxx	1111	1111 以外	レジスタロード符号付きハーフワード	P.A6-129「*LDRSH（リテラル）*」
10	000000	1111 以外	1111 以外	レジスタロード符号付きハーフワード	P.A6-131「*LDRSH（レジスタ）*」
xx	xxxxxx	xxxxxx	1111	未割り当てメモリヒント[a]	-

a. 未割り当てメモリヒントは NOP として実装する必要があります。ソフトウェアでは使用しないで下さい。

A5.3.9 バイトのロード、メモリヒント

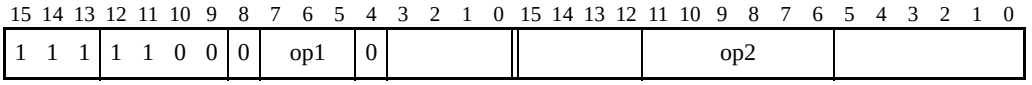


この空間でのエンコードの割り当てを、表 A5-20 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-20 バイトのロード、プリロード

op1	op2	Rn	Rt	命令	参照先
01	xxxxxx	1111 以外	1111 以外	レジスタロード バイト	P.A6-95「*LDRB（イミディエート）*」
00	1xx1xx	1111 以外			
00	1100xx	1111 以外	1111 以外		
00	1110xx	1111 以外		非特権レジスタロード バイト	P.A6-101「*LDRBT*」
0x	xxxxxx	1111	1111 以外	レジスタロード バイト	P.A6-97「*LDRB（リテラル）*」
00	000000	1111 以外	1111 以外	レジスタロード バイト	P.A6-99「*LDRB（レジスタ）*」
11	xxxxxx	1111 以外	1111 以外	レジスタロード符号付き バイト	P.A6-119「*LDRSB（イミディエート）*」
10	1xx1xx	1111 以外			
10	1100xx	1111 以外	1111 以外		
10	1110xx	1111 以外		非特権レジスタロード符号付きバイト	P.A6-125「*LDRSBT*」
1x	xxxxxx	1111	1111 以外	レジスタロード符号付き バイト	P.A6-121「*LDRSB（リテラル）*」
10	000000	1111 以外	1111 以外	レジスタロード符号付き バイト	P.A6-123「*LDRSB（レジスタ）*」
01	xxxxxx	1111 以外	1111	データのプリロード	P.A6-177「*PLD（イミディエート、リテラル）*」
00	1100xx	1111 以外	1111		
0x	xxxxxx	1111	1111		
00	000000	1111 以外	1111	データのプリロード	P.A6-179「*PLD（レジスタ）*」
11	xxxxxx	1111 以外	1111	命令のプリロード	P.A6-181「*PLI（イミディエート、リテラル）*」
10	1100xx	1111 以外	1111		
1x	xxxxxx	1111	1111		
10	000000	1111 以外	1111	命令のプリロード	P.A6-183「*PLI（レジスタ）*」

A5.3.10 単一データ項目のストア

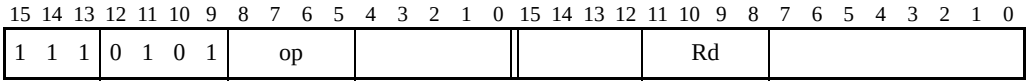


この空間でのエンコードの割り当てを、表 A5-21 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-21 単一データ項目のストア

op1	op2	命令	参照先
100	xxxxxx	レジスタストア バイト	P.A6-225「*STRB（イミディエート）*」
000	1xxxxx		
000	0xxxxx	レジスタストア バイト	P.A6-227「*STRB（レジスタ）*」
101	xxxxxx	レジスタストア ハーフワード	P.A6-237「*STRH（イミディエート）*」
001	1xxxxx		
001	0xxxxx	レジスタストア ハーフワード	P.A6-239「*STRH（レジスタ）*」
110	xxxxxx	レジスタストア	P.A6-221「*STR（イミディエート）*」
010	1xxxxx		
010	0xxxxx	レジスタストア	P.A6-223「*STR（レジスタ）*」

A5.3.11 データ処理（シフトしたレジスタ）



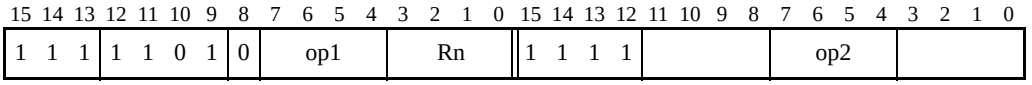
この空間でのエンコードの割り当てを、表 A5-22 に示します。

この空間に含まれる他のエンコードは未定義です。

表 A5-22 データ処理（シフトしたレジスタ）

op	Rd	命令	参照先
0000	1111 以外	ビット単位論理積	P.A6-34「*AND（レジスタ）*」
0000	1111	テスト	P.A6-263「*TST（レジスタ）*」
0001		ビットクリア	P.A6-46「*BIC（レジスタ）*」
0010	1111 以外	論理和	P.A6-175「*ORR（レジスタ）*」
0010	1111	移動	P.A6-151「*MOV（レジスタ）*」
0011	1111 以外	否定論理和	P.A6-171「*ORN（レジスタ）*」
0011	1111	ビット単位否定	P.A6-165「*MVN（レジスタ）*」
0100	1111 以外	排他的論理和	P.A6-74「*EOR（レジスタ）*」
0100	1111	等価テスト	P.A6-260「*TEQ（レジスタ）*」
1000	1111 以外	加算	P.A6-24「*ADD（レジスタ）*」
1000	1111	比較否定	P.A6-60「*CMN（レジスタ）*」
1010		キャリー付き加算	P.A6-20「*ADC（レジスタ）*」
1011		キャリー付き減算	P.A6-205「*SBC（レジスタ）*」
1101	1111 以外	減算	P.A6-245「*SUB（レジスタ）*」
1101	1111	比較	P.A6-64「*CMP（レジスタ）*」
1110		逆減算	P.A6-201「*RSB（レジスタ）*」

A5.3.12 データ処理（レジスタ）



命令の 2 番目のハーフワードがビット [15:12] != 0b1111 の場合、その命令は未定義です。

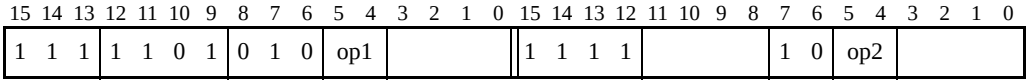
この空間でのエンコードの割り当てを、表 A5-23 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-23 データ処理（レジスタ）

op1	op2	命令	参照先
000x	0000	論理左シフト	P.A6-137「*LSL（レジスタ）*」
001x	0000	論理右シフト	P.A6-141「*LSR（レジスタ）*」
010x	0000	算術右シフト	P.A6-38「*ASR（レジスタ）*」
011x	0000	右ローテート	P.A6-195「*ROR（レジスタ）*」
0000	1xxx	符号付きハーフワード拡張	P.A6-255「*SXTH*」[a]
0001	1xxx	符号なしハーフワード拡張	P.A6-273「*UXTH*」[a]
0100	1xxx	符号付きバイト拡張	P.A6-253「*SXTB*」[a]
0101	1xxx	符号なしバイト拡張	P.A6-273「*UXTH*」[a]
10xx	10xx	P.A5-28「*その他の操作*」参照	

a. この場合、Rn == 1111。

A5.3.13 その他の操作

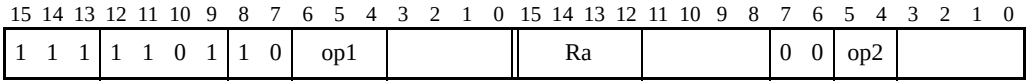


命令の 2 番目のハーフワードがビット [15:12] != 0b1111 の場合、その命令は未定義です。
この空間でのエンコードの割り当てを、表 A5-24 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-24 その他の操作

op1	op2	命令	参照先
01	00	ワードのバイト反転	P.A6-190「*REV*」
01	01	パックハーフワードのバイト反転	P.A6-191「*REV16*」
01	10	ビット反転	P.A6-189「*RBIT*」
01	11	符号付きハーフワードのバイト反転	P.A6-192「*REVSH*」
11	00	先行ゼロカウント	P.A6-57「*CLZ*」

A5.3.14 乗算、積和演算、および絶対値の差



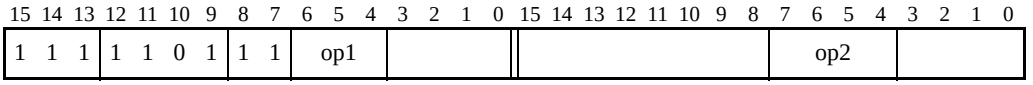
命令の 2 番目のハーフワードがビット [7:6] != 0b00 の場合、その命令は未定義です。

この空間でのエンコードの割り当てを、表 A5-25 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-25 乗算、積和演算、および絶対値の差の操作

op1	op2	Ra	命令	参照
000	00	1111 以外	積和演算	P.A6-147「*MLA*」
000	00	1111	乗算	P.A6-161「*MUL*」
000	01		積減算	P.A6-148「*MLS*」

A5.3.15 ロング乗算、ロング積和演算、および除算

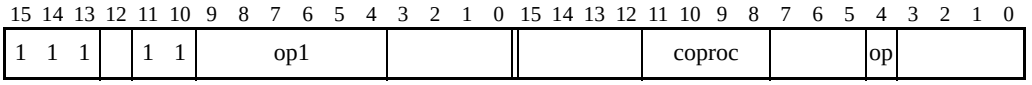


この空間でのエンコードの割り当てを、表 A5-26 に示します。この空間に含まれる他のエンコードは未定義です。

表 A5-26 乗算、積和演算、および絶対値の差の処理

op1	op2	命令	参照先
000	0000	符号付きロング乗算	P.A6-213「*SMULL*」
001	1111	符号付き除算	P.A6-209「*SDIV*」
010	0000	符号なしロング乗算	P.A6-268「*UMULL*」
011	1111	符号なし除算	P.A6-266「*UDIV*」
100	0000	符号付きロング積和演算	P.A6-212「*SMLAL*」
110	0000	符号なしロング積和演算	P.A6-267「*UMLAL*」

A5.3.16 コプロセッサ命令



この空間でのエンコードの割り当てを、表 A5-27 に示します。この空間に含まれる他のエンコード、および対象のコプロセッサが存在しない場合は未定義です。

表 A5-27 コプロセッサ命令

op1	op	coproc	命令	参照先
0xxxx0[a]	x	xxxx	コプロセッサストア	P.A6-215「*STC*、*STC2*」
0xxxx1[a]	x	xxxx	コプロセッサロード	P.A6-80「*LDC*、*LDC2（イミディエート）*」, P.A6-82「*LDC*、*LDC2（リテラル）*」
000100	x	xxxx	2 つの ARM コアレジスタからコプロセッサへの移動	P.A6-145「*MCRR, MCRR2*」
000101	x	xxxx	コプロセッサから 2 つの ARM コアレジスタへの移動	P.A6-157「*MRRC*、*MRRC2*」
10xxxx	0	xxxx	コプロセッサデータ演算	P.A6-54「*CDP, CDP2*」
10xxx0	1	xxxx	ARM コアレジスタからコプロセッサへの移動	P.A6-143「*MCR*、*MCR2*」
10xxx1	1	xxxx	コプロセッサから ARM コアレジスタへの移動	P.A6-155「*MRC, MRC2*」

a. ただし、000x0x 以外

第 A6 章
Thumb 命令の詳細

本章では、ARMv7-M での Thumb-2® 命令サポートについて説明します。本章は以下のセクションから構成されています。

A6.1 命令の説明フォーマット

P.A6-17「*ARMv7-M Thumb 命令のアルファベット順リスト*」に記載されている命令のアルファベット順リストの説明では、特別な場合を除いて次のフォーマットが使用されます。

- 命令セクションのタイトル
- 命令の紹介
- 命令のエンコードとアーキテクチャ情報
- アセンブラ構文
- 命令の動作を説明する擬似コード
- 例外情報
- 注（該当する場合）

それぞれの項目について、以後のサブセクションで詳しい説明が行われます。

命令の説明によっては、他の命令の代替ニーモニックが紹介されたり、このフォーマットの省略バージョンや変更されたバージョンが使用されたりする場合があります。

A6.1.1 命令セクションのタイトル

命令セクションのタイトルでは、そのセクションで説明する命令の基本のニーモニックが示されます。ひとつのニーモニックが別の命令セクションで記述される複数の形式を持つ場合、ニーモニックの後に括弧内でその形式の簡単な説明が示されます。この説明は主に、オペランドの 1 つがイミディエート値である形式と、レジスタを使用する形式を区別するために使用されます。

新しいアセンブラ構文でニーモニックが完全に別のものに置き換えられている場合、以前のニーモニックが括弧内に示されます。

A6.1.2 命令の紹介

命令セクションのタイトルの次に、その命令の主な機能に関する短い説明が続きます。この説明は、必ずしも完全ではなく、命令の明確な定義ではありません。この説明とその後に続く詳細情報が矛盾する場合は、後者が優先されます。

A6.1.3 命令のエンコード

「エンコード」サブセクションには、1 つ以上の命令のエンコードのリストが含まれます。各 Thumb 命令のエンコードには、参照用に T1、T2、T3 などのラベルが付けられています。

命令のエンコードの説明には、次の項目が含まれています。

- アセンブラがそのエンコードを他のどのエンコードよりも優先的に選択することが保証されているアセンブラ構文。複数の構文が示される場合があります。使用すべき適切な構文が、IT ブロック内部や IT ブロック外部などの構文に付加された注釈によって示されている場合もあります。それ以外の場合では、アセンブラ構文の説明を読んで、逆アセンブルした命令がどの構文に対応するかを理解することによって、適切な構文を判定できます。

  通常は、何らかの特定のエンコーディングに再アセンブルできる構文が複数存在します。このような構文の正確なセットは、一般にレジスタ番号、イミディエート定数、および命令に対するその他のオペランドによって異なります。例えば、Thumb 命令セットにアセンブルする場合、`AND R0,R0,R8` という構文では 32 ビットエンコードが必ず選択されますが、`AND R0,R0,R1` という構文では 16 ビットエンコードが選択されます。

  エンコード用に記述されているアセンブラ構文は、そのエンコードでサポートされているすべてのオペランドの組み合わせの中で、最も単純なものが選択されています。このため多くの場合、そのアセンブラ構文には、オペランドの組み合わせのうちの一部に対してのみ必要な要素が含まれています。例えば、32 ビット Thumb の AND（レジスタ）エンコードについて記載されているアセンブラ構文では、一部のオペランドの組み合わせについては 16 ビットエンコードが使用可能ですが、そのような組み合わせについても、必ず 32 ビットエンコードが選択されるように、.W 修飾子が使用されています。

  したがって、あるエンコードに対して付加されたアセンブラ構文は、そのエンコードを逆アセンブルするための逆アセンブラでの使用にも適しています。ただし、逆アセンブラでは、逆アセンブルされたコードを読みやすくするため、特定のオペランドの組み合わせに適した、より単純な構文を使用してもかまいません。

- エンコード図。16 ビット Thumb エンコードの場合は、32 ビット Thumb エンコードの場合の半分の幅で示されます。P.A3-7「*命令アライメントおよびバイトの順序付け*」で説明されているように、32 ビット Thumb エンコードでは、ワードではなく、2 つのハーフワードのシーケンスのバイト順序が使用されることを示すために、2 つのハーフワードの間に縦の二重線が示されています。
- エンコード固有の擬似コード。これは、エンコード固有の命令フィールドを、後述の「*動作*」サブセクション内のエンコード独立の擬似コードへの入力に変換し、エンコードの特殊な条件を抽出する擬似コードです。使用される擬似コードの詳細と、エンコード図、エンコード固有の擬似コード、およびエンコード独立の擬似コードとの関係については、付録 D *「擬似コードの定義」*を参照して下さい。

A6.1.4 アセンブラ構文

「アセンブラ構文」サブセクションには、その命令について標準的な UAL 構文が記載されています。

各構文の説明には、次の項目が含まれています。

- P.A6-4「*アセンブラ構文プロトタイプ行の規則*」で説明されている規則を使用して、`typewriter` フォントで記載された 1 つ以上の構文プロトタイプ行。各プロトタイプ行には、一行全体のアセンブラコードに含まれる、ニーモニックとオペランドの部分（該当する場合）が記載されています。このような行が複数ある場合は、各プロトタイプ行に、エ

ンコード固有の擬似コードに必要な結果を示す注釈が付けられます。命令エンコーディングのそれぞれについて、その構文をアセンブルする際にそのエンコーディングに適合する何からの命令があるか、もしあるならどれかを決定するのに使うことができます。

- プロトタイプ構文行のすべての変数またはオプションフィールドの説明が続く「*各項目の説明については以下を参照して下さい*」行。

  一部の構文フィールドは、すべての命令またはほとんどの命令について標準化されています。このようなフィールドについては、P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

  デフォルトで、レジスタを指定する構文フィールド（<Rd>、<Rn>、<Rt> など）は、Thumb 命令内で R0 ～ R12 または LR のいずれかに置き換えることができます。この場合は、エンコード固有の擬似コードで、整数変数（d、n、t など）を対応するレジスタ番号（R0 ～ R12 の場合は 0 ～ 12、LR の場合は 14）に設定する必要があります。通常は、命令内の対応するビットフィールド（Rd、Rn、Rt などの名前が付けられています）を、その番号のバイナリエンコードに設定することによって実現されます。16 ビット Thumb エンコードの場合は、このビットフィールドの長さが 3 であるため、アセンブラ構文で R0 ～ R7 のいずれかが指定されている場合にのみこのエンコードが使用できます。また、このようなエンコードでは、Rdn などのビットフィールド名が多く使用されます。これは、<<Rd> と <Rn> が同じレジスタを指している場合にのみこのエンコードが使用でき、その場合は、そのレジスタの番号がビットフィールドにエンコードされることを意味しています。

  レジスタを指定する構文フィールドの説明では、レジスタの許容範囲が拡張または制限されていることや、その他フィールドのデフォルトの規則とは相違点があることが記載されている場合があります。このような拡張の例として、SP や PC が使用可能である場合が挙げられます（それぞれ、レジスタ番号の 13 と 15 を使用します）。

注

UAL 以前の Thumb アセンブラ構文は、UAL と互換性がないため、命令セクションでは使用されていません。

アセンブラ構文プロトタイプ行の規則

アセンブラ構文プロトタイプ行とそのサブフィールドには、次の規則が適用されます。

< >　< と > で囲まれた項目は、ユーザがその位置に与える必要のある値のタイプに関する短い説明です。通常、その項目に関するより詳しい説明が後に続きます。このような項目の多くは、命令のエンコード図で同様の名前が付けられたフィールドに対応しています。この対応で、命令のエンコードに代入する整数値またはレジスタ番号のバイナリエンコードだけが必要な場合、その説明は省略されます。例えば、Thumb 命令のアセンブラ構文に <Rn> という項目が含まれており、命令のエンコード図に Rn という名前の 4 ビットフィールドが含まれている場合は、アセンブラ構文で指定されたレジスタの番号がバイナリで命令フィールドにエンコードされます。

アセンブラ構文の項目と命令のエンコードの間の対応が、整数やレジスタ番号の単純なバイナリエンコードよりも複雑な場合は、その項目の説明でどのようにエンコードするかが示されています。多くの場合、エンコード固有の擬似コードからの必須の出力を add =TRUE などで指定することによって実現されます。アセンブラでは、この出力を生成するエンコード以外は使用しないようにする必要があります。

{ } { と } で囲まれた項目はオプションです。通常、この項目の説明と、この項目が存在する場合と存在しない場合に命令がどのようにエンコードされるかの説明が後に続きます。

多くの命令で、オプションとしてデスティネーションレジスタが使用されます。特記されていない限り、デスティネーションレジスタが省略されている場合は、命令構文で直後にあるソースレジスタと同一のレジスタが使用されます。

スペース　項目を分離してわかりやすくするために単一のスペースが使用されます。アセンブラ構文でスペースが必須の場合は、連続する複数のスペースが使用されます。

+/−　オプションの +/- 符号を示しています。コードにどちらも指定されていない場合、+ と見なされます。

他のすべての文字は、アセンブラ構文に記載されているのと正確に同じにエンコードする必要があります。本書内のアセンブラ命令の基本形式には、{ および } を除いて、上に説明されている特殊文字は記載されていません。{ および } 文字は、一部の場所で、変数項目の一部としてエンコードする必要があります。これらの文字が使用されている部分では、変数項目の説明でそれらの使用方法が示されます。

A6.1.5 命令の動作を説明する擬似コード

「*動作*」サブセクションには、その命令の主要な動作を説明する、エンコード独立の擬似コードが含まれています。使用されている擬似コードの詳細と、エンコード図、エンコード固有の擬似コード、およびエンコード独立の擬似コード間の関係については、付録 D *「擬似コードの定義」*を参照して下さい。

A6.1.6 例外情報

「*例外*」サブセクションには、命令の実行によって発生する可能性のある例外状態のリストが含まれています。

プロセッサ例外は、次の規則に従って記載されています。

- リセットと割り込み（NMI、PendSV、SysTick を含む）は記載されていません。これらは、任意の命令の実行前または実行後、および特定の命令の実行中に発生する可能性がありますが、命令によって発生することは通常はありません。
- MemManage 例外と BusFault 例外は、明示的なデータメモリアクセスを実行するすべての命令に対して記載されています。

  MemManage 例外と BusFault 例外は、すべての命令フェッチで発生の可能性があります。命令の実行では発生しないため、記載されていません。
- UsageFault 例外は、さまざまな理由で発生する可能性があり、該当する命令に記載されています。

  また、UsageFault 例外は、その命令が擬似コードで未定義命令であることが示されている場合にも発生します。このような UsageFault 例外は記載されていません。
- SVC 例外は、SVC 命令に対して記載されています。
- DebugMonitor 例外は、BKPT 命令に対して記載されています。
- HardFault 例外は、命令に記載されているフォルトの昇格によって発生する可能性がありますが、単独では記載されていません。

注

MemManage 例外、BusFault 例外、UsageFault 例外の各タイプの概要については、P.B1-18「*フォルトの動作*」を参照して下さい。

A6.1.7 注

必要に応じて、小見出しの下に命令に関する注記が記載されています。

A6.2 標準アセンブラ構文のフィールド

次のアセンブラ構文フィールドは、ほぼすべての命令で共通しています。

<c> オプションフィールドで、この命令が実行される条件を指定します。<c> が省略されている場合、デフォルトで*常時* (AL) に設定されます。詳細については、P.A4-3「*条件付き実行*」を参照して下さい。

<q> 命令に関するオプションのアセンブラ修飾子を指定します。次の修飾子が定義されています。

.N Narrow（狭い）を意味し、その命令に対してアセンブラが 16 ビットエンコードを選択する必要があることを指定します。これが不可能な場合は、アセンブラエラーが発生します。

.W Wide（広い）を意味し、その命令に対してアセンブラが 32 ビットエンコードを選択する必要があることを示します。これが不可能な場合は、アセンブラエラーが発生します。

.W と .N のどちらも指定されていない場合、アセンブラが 16 ビットエンコードと 32 ビットエンコードのいずれかを選択します。両方が使用可能な場合、アセンブラは 16 ビットエンコードを選択する必要があります。ひとつの命令に対して同じ長さのエンコーディングが複数利用可能な場合、どのエンコードを選択するかの規則は命令によって異なり、命令の説明で示されています。

A6.3 条件付き実行

ほとんどの Thumb-2 命令は、APSR 条件フラグの値に基づいて、条件付きで実行することができます。使用可能な条件のリストを表 A6-1 に示します。

Thumb 命令では、条件（AL 以外の場合）は一般に、先行する IT 命令にエンコードされます。詳細については、P.A4-4「*条件付き命令*」、P.A6-10「*ITSTATE*」、および P.A6-78「*IT*」を参照して下さい。条件分岐命令によっては、エンコードに条件コードが含まれており、前の IT 命令が必要ないものもあります。

表 A6-1 条件コード

cond	ニーモニック拡張	意味（整数）	意味（浮動小数点）[a]	条件フラグ
0000	EQ	等しい	等しい	Z == 1
0001	NE	等しくない	等しくない、または規則なし	Z == 0
0010	CS [b]	キャリーセット	≧、または規則なし	C == 1
0011	CC [c]	キャリークリア	＜	C == 0
0100	MI	マイナス / 負	＜	N == 1
0101	PL	プラス、正、またはゼロ	≧、または規則なし	N == 0
0110	VS	オーバフロー	規則なし	V == 1
0111	VC	オーバフローなし	規則あり	V == 0
1000	HI	符号なし＞	＞、または規則なし	C == 1 and Z == 0
1001	LS	符号なし≦	≦	C == 0 or Z == 1
1010	GE	符号付き≧	≧	N == V
1011	LT	符号付き＜	＜、または規則なし	N != V
1100	GT	符号付き＞	＞	Z == 0 and N == V
1101	LE	符号付き≦	≦、または規則なし	Z == 1 or N != V
1110	なし (AL) [d]	常時（無条件）	常時（無条件）	Any

a. 規則なしは、少なくとも 1 つは NaN オペランドであることを意味します。
b. HS（符号なし≧）は、CS と同義語です。
c. LO（符号なし＜）は、CC と同義語です。
d. AL は、IT 命令を除いて、常時を意味するオプションのニーモニック拡張です。詳細については、P.A6-78「*IT*」を参照して下さい。

A6.3.1 条件付き実行の擬似コードの詳細

CurrentCond() 擬似コード関数には次のプロトタイプがあります。

```
bits(4) CurrentCond()
```

このプロトタイプは、次の 4 ビットの条件指定子を返します。

- 分岐命令（P.A6-40「*B*」参照）の T1 および T3 エンコードでは、4 ビットの cond フィールドを返します。
- 他のすべての Thumb 命令では、ITSTATE.IT[7:4] を返します。詳細については、P.A6-10「*ITSTATE*」を参照して下さい。

ConditionPassed() 関数は、この条件指定子と APSR 条件フラグを使用して、命令を実行すべきかどうかを判断します。

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    cond = CurrentCond();

    // Evaluate base condition.
    case cond<3:1> of
        when '000'  result = (APSR.Z == '1');                       // EQ or NE
        when '001'  result = (APSR.C == '1');                       // CS or CC
        when '010'  result = (APSR.N == '1');                       // MI or PL
        when '011'  result = (APSR.V == '1');                       // VS or VC
        when '100'  result = (APSR.C == '1') && (APSR.Z == '0');    // HI or LS
        when '101'  result = (APSR.N == APSR.V);                    // GE or LT
        when '110'  result = (APSR.N == APSR.V) && (APSR.Z == '0'); // GT or LE
        when '111'  result = TRUE;                                  // AL

    // Condition bits '111x' indicate the instruction is always executed. Otherwise,
    // invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

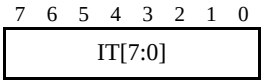
A6.3.2 未定義命令の条件付き実行

未定義命令が ARMv7-M の条件チェックで条件が成立しなかった場合、その命令は NOP として動作し、例外は発生しません。

注

cond フィールドが "1110" の分岐 (B) 命令は未定義であり、IT 命令の条件チェックで条件が成立すると、例外が発生します。

A6.3.3 ITSTATE



このフィールドには、Thumb IT 命令の If-Then 実行状態ビットが保持されます。IT 命令と関連する IT ブロックの説明については、P.A6-78「*IT*」を参照して下さい。

ITSTATE は、次の 2 つのサブフィールドで構成されます。

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

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

IT[4:0] 次のようにエンコードされます。

- IT ブロックのサイズ。これは、条件付きで実行する命令の数です。P.A6-11 表 A6-2 に示されているように、ブロックのサイズは、このフィールド内の最下位の 1 の位置によって表されます。
- ブロック内の各命令の条件コード内で最下位ビットの値

注

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

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

IT 命令が実行されると、命令内の条件と、*Then* および *Else*（T および E）パラメータに従って、これらのビットがセットされます。詳細については、P.A6-78「*IT*」を参照して下さい。

IT ブロック内部の命令は条件付きです。詳細については、P.A4-4「*条件付き命令*」を参照して下さい。使用される条件は、IT[7:4] の現在の値です。IT ブロック内部の 1 つの命令が正常終了すると、ITSTATE は P.A6-11 表 A6-2 の次の行に移行します。

このような命令で例外が取得された場合の動作の詳細については、P.B1-15「*例外の開始*」を参照して下さい。

注

正常終了時に分岐を引き起こす可能性がある命令は、IT ブロック内部の最後の命令にのみ許可されます。これによって、ITSTATE はその命令の実行後に必ず通常の実行に移行します。

表 A6-2 IT 実行状態ビットの効果

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

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

ITSTATE 動作の擬似コードの詳細

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

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

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

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

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

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

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

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

A6.4 レジスタに適用されるシフト

ARM レジスタオフセットを使用する、ワードおよび符号なしバイトのロード / ストア命令では、各種の定数シフトをオフセットレジスタに適用することができます。Thumb-2 および ARM のデータ処理命令では、同じ範囲の各種の定数シフトを 2 番目のオペランドレジスタに適用することができます。詳細については、*定数シフト*を参照して下さい。

ARM データ処理命令は、レジスタ制御シフトを 2 番目のオペランドレジスタに適用することができます。

A6.4.1 定数シフト

定数シフトは、入力ビットが異なる位置からくること以外は、Thumb-2 命令と ARM 命令とで同じです。

<shift> は、<Rm> に適用されるオプションのシフトです。次のいずれかのシフトを使用できます。

（省略） LSL #0 と等価

LSL #<n> <n> ビットの論理左シフト (0 <=<n><= 31)

LSR #<n> <n> ビットの論理右シフト (1<n><= 32)

ASR #<n> <n> ビットの算術右シフト (1 <=<n><= 32)

ROR #<n> <n> ビットの右ローテート (1 <=<n><= 31)

RRX 拡張付き 1 ビット右ローテート。ビット [0] を shifter_carry_out に書き込み、ビット [31:1] を右に 1 ビットシフトし、キャリーフラグをビット [31] にシフトします。

エンコード

アセンブラは、次に示すように <shift> を 2 つの type ビットと、5 つの immediate ビットにエンコードします。

（省略） type = 0b00、immediate = 0

LSL #<n> type = 0b00、immediate = <n>

LSR #<n> type = 0b01
<n>< 32 の場合、immediate = <n>
<n>== 32 の場合、immediate = 0

ASR #<n> type = 0b10
<n>< 32 の場合、immediate =<n>
<n>== 32 の場合、immediate = 0

ROR #<n> type = 0b11、immediate = <n>

RRX type = 0b11、immediate = 0

A6.4.2 レジスタ制御シフト

このシフトは、ARM 命令でのみ使用できます。

<type> は<Rm> から読み出された値に適用されるシフトのタイプです。次のいずれかを使用できます。

ASR 算術右シフト。type = 0b10 としてエンコードされます。

LSL 論理左シフト。type = 0b00 としてエンコードされます。

LSR 論理右シフト。type = 0b01 としてエンコードされます。

ROR 右ローテート。type = 0b11 としてエンコードされます。

<Rs> の下位バイトには、シフト量が含まれています。

A6.4.3 シフト操作

```
// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

    case type of
        when '00'
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);

// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) type)
    case type of
        when '00'  shift_t = SRType_LSL;
        when '01'  shift_t = SRType_LSR;
        when '10'  shift_t = SRType_ASR;
        when '11'  shift_t = SRType_ROR;
    return shift_t;


// Shift()
// =======

bits(N) Shift(bits(N) value, SRType type, integer amount, bit carry_in)
    (result, -) = Shift_C(value, type, amount, carry_in);
    return result;
```



```
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```


A6.5 メモリアクセス

次のアドレシングモードは、メモリアクセス命令で共通に使用することができます。

オフセットアドレシング

ベースレジスタから取得されたアドレスに対してオフセット値が加算または減算され、結果がメモリアクセスのアドレスとして使用されます。ベースレジスタは変更されません。

このモードのアセンブリ言語構文は次のとおりです。

[<Rn>,<offset>]

プリインデクスアドレシング

ベースレジスタから取得されたアドレスにオフセット値が適用され、結果がメモリアクセスのアドレスとして使用され、ベースレジスタに書き戻されます。

このモードのアセンブリ言語構文は次のとおりです。

[<Rn>,<offset>]!

ポストインデクスアドレシング

ベースレジスタから取得されたアドレスが、そのままメモリアクセス用のアドレスとして使用されます。オフセット値がアドレスに適用され、結果がベースレジスタに書き戻されます。

このモードのアセンブリ言語構文は次のとおりです。

[<Rn>],<offset>

それぞれのケースで、<Rn> はベースレジスタを表します。<offset> には次のものが使用できます。

- <imm8> や <imm12> などのイミディエート定数
- インデクスレジスタ <Rm>
- <Rm>, LSL #<shift> などのシフトしたインデクスレジスタ

アンアラインドアクセス、エンディアン形式、排他アクセスの詳細については、次の項を参照して下さい。

- *アライメントのサポート*：P.A3-3
- *エンディアンのサポート*：P.A3-5
- *同期化とセマフォ*：P.A3-9

A6.6 ヒント命令

Thumb ISA には、次の 2 つのクラスのヒント命令があります。

- メモリヒント
- NOP 互換ヒント

A6.6.1 メモリヒント

Rt == 0b1111 である一部のロード命令はメモリヒントです。メモリヒントを使用すると、データの実際のロード/ストアを行わずに将来のメモリアクセスに関する事前情報をメモリシステムに提供することができます。

現在定義されているメモリヒント命令は PLD および PLI のみです。P.A5-24「*バイトのロード、メモリヒント*」を参照して下さい。命令の詳細については、次の項を参照して下さい。

- *PLD（イミディエート、リテラル）*: P.A6-177
- *PLD（レジスタ）*: P.A6-179
- *PLI（イミディエート、リテラル）*: P.A6-181
- *PLI（レジスタ）*: P.A6-183

その他のメモリヒントは、現在割り当てられていません。詳細については、P.A5-23「*ハーフワードのロード、未割り当てメモリヒント*」を参照して下さい。メモリヒント命令の効果は実装定義です。未割り当てのメモリヒントは、NOP として実装する必要があり、ソフトウェアでは使用しないで下さい。

A6.6.2 NOP 互換ヒント

メモリアクセスに関連付けられていないヒント命令は、NOP 互換ヒントと呼ばれる、ヒント命令の別のカテゴリに分類されます。NOP 互換ヒントは、実装定義の動作を提供するか、NOP として機能します。16 ビットエンコードと 32 ビットエンコードの両方が予約されています。

- 16 ビットエンコードの詳細については、P.A5-11「*If-Then とヒント*」を参照して下さい。
- 32 ビットエンコードの詳細については、P.A5-19「*ヒント命令*」を参照して下さい。

A6.7 ARMv7-M Thumb 命令のアルファベット順リスト

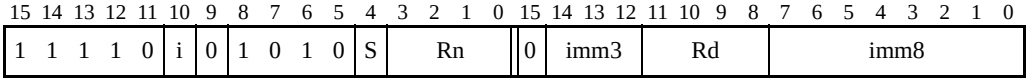
このセクションには、すべての ARMv7-M Thumb 命令が記載されています。使用されているフォーマットについては、P.A6-2「*命令の説明フォーマット*」を参照して下さい。

A6.7.1 ADC（イミディエート）

ADC（キャリー付き加算（イミディエート））は、イミディエート値とキャリーフラグの値をレジスタの値に加算し、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

ADC{S}<c> <Rd>,<Rn>,#<const>




```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```


アセンブラ構文

```
ADC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

〈c〉〈q〉　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

〈Rd〉　　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

〈Rn〉　　第一オペランドを格納しているレジスタを指定します。

〈const〉　　<Rn> から取得される値に加算されるイミディエート値を指定します。許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

UAL 以前の構文の ADC<c>S は、ADCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

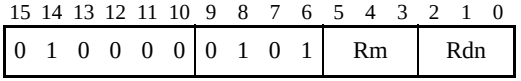
A6.7.2 ADC（レジスタ）

ADC（キャリー付き加算（レジスタ））は、レジスタの値、キャリーフラグの値、およびオプションで、シフトされたレジスタの値を加算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

ADCS <Rdn>,<Rm> IT ブロック外部

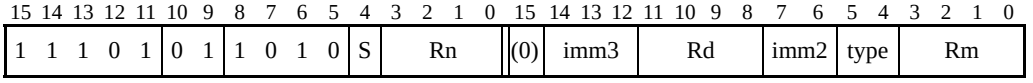
ADC<c> <Rdn>,<Rm> IT ブロック内部




```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

ADC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
ADC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　2 番目のオペランドとして使用されるレジスタを指定します。オプションとして、このレジスタの値をシフトできます。

<shift>　　<Rm> から読み出される値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われず、両方のエンコードが許可されます。<shift> を指定した場合、エンコード T2 のみが許可されます。どのようなシフトが可能か、およびそれらのエンコード方法については、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

特殊なケースとして、どちらもR0～R7の範囲に含まれる<Rd>と<Rn>を使用して ADC<c> <Rd>,<Rn>,<Rd> と記述した場合は、ADC<c> <Rd>,<Rn> と記述されたものとして、エンコード T2 を使用してアセンブルされます。これを避けるには、.W 修飾子を使用します。

UAL 以前の構文の ADC<c>S は、ADCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

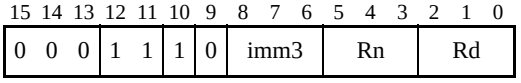
A6.7.3 ADD（イミディエート）

ADD（加算（イミディエート））は、イミディエート値をレジスタの値に加算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

ADDS <Rd>,<Rn>,#<imm3> IT ブロック外部

ADD<c> <Rd>,<Rn>,#<imm3> IT ブロック内部

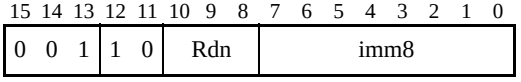



```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

ADDS <Rdn>,#<imm8> IT ブロック外部

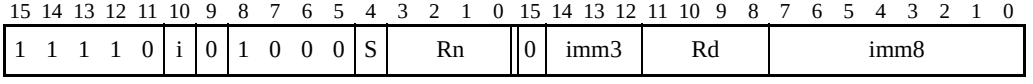
ADD<c> <Rdn>,#<imm8> IT ブロック内部




```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

エンコード T3 ARMv6T2、ARMv7

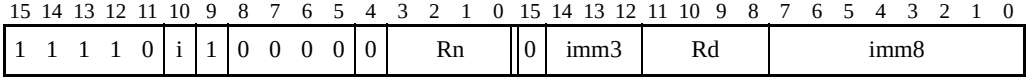
ADD{S}<c>.W <Rd>,<Rn>,#<const>




```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || n == 15 then UNPREDICTABLE;
```

エンコード T4 ARMv6T2、ARMv7

ADDW<c> <Rd>,<Rn>,#<imm12>




```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if BadReg(d) then UNPREDICTABLE;
```


アセンブラ構文

ADD{S}<c><q> {<Rd>,} <Rn>, #<const>	すべてのエンコードが許可される
ADDW<c><q> {<Rd>,} <Rn>, #<const>	エンコード T4 のみが許可される

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> と同じレジスタが使用されます。

<Rn>　第一オペランドを格納しているレジスタを指定します。<Rn> に SP を指定した場合は、P.A6-26「*ADD (SP + イミディエート)*」を参照して下さい。<Rn> に PC を指定した場合は、P.A6-30「*ADR*」を参照して下さい。

<const>　<Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲は、エンコード T1 では 0 ～ 7、エンコード T2 では 0 ～ 255、エンコード T4 では 0 ～ 4095 です。T3 エンコードで許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

同じ長さの複数のエンコードが使用可能な場合は、エンコード T4 よりもエンコード T3 の方が適しています（エンコード T4 が必要な場合は、ADDW 構文を使用して下さい）。<Rd> を指定する場合は、エンコード T2 よりもエンコード T1 の方が適しています。<Rd> を省略する場合は、エンコード T1 よりもエンコード T2 の方が適しています。

UAL 以前の構文の ADD<c>S は、ADDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

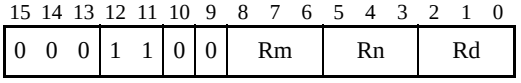
A6.7.4 ADD（レジスタ）

ADD（加算（レジスタ））は、レジスタ値と、オプションでシフトしたレジスタ値を加算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv7

ADDS <Rd>,<Rn>,<Rm> IT ブロック外部

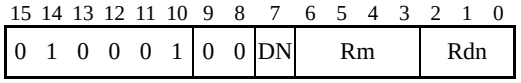
ADD<c> <Rd>,<Rn>,<Rm> IT ブロック内部




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

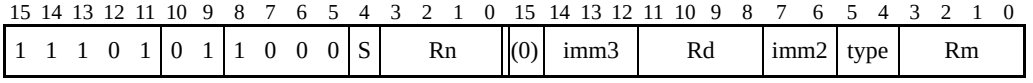
ADD<c> <Rdn>,<Rm>




```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn);  n = UInt(DN:Rdn);  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if d == 15 && m == 15 then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

ADD{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}




```
if Rd == '1111' && S == '1' then SEE CMN (register);
if Rn == '1101' then SEE ADD (SP plus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || n == 15 || BadReg(m) then UNPREDICTABLE;
```


アセンブラ構文

ADD{S}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> と同じレジスタが使用されます。両方が使用可能な場合、エンコード T1 よりもエンコード T2 の方が適しています（IT ブロック内部でのみ可能性があります）。<Rd> を指定した場合は、エンコード T2 よりもエンコード T1 の方が適しています。

<Rn>　最初のオペランドを含むレジスタを指定します。<Rn> に SP を指定した場合については、P.A6-28「*ADD (SP + レジスタ)*」を参照して下さい。

<Rm>　オプションでシフトされた、2 番目のオペランドとして使用するレジスタを指定します。

<shift>　<Rm> から読み出された値に適用するシフトを指定します。<shift> を省略した場合、シフトは行われず、すべてのエンコードが許可されます。<shift> を指定した場合、エンコード T3 のみが許可されます。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

IT ブロック内部で、エンコード T1 を使用して ADD<c> <Rd>,<Rn>,<Rd> をアセンブルできない場合、ADD<c> <Rd>,<Rn> と記述されたものとしてエンコード T2 を使用してアセンブルされます。これを避けるには、.W 修飾子を使用します。

UAL 以前の構文の ADD<c>S は、ADDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

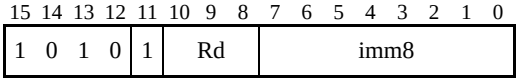
なし

A6.7.5 ADD（SP + イミディエート）

ADD（加算（SP + イミディエート））は、イミディエート値を SP の値に加算して、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

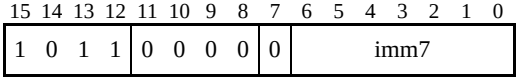
ADD<c> <Rd>,SP,#<imm8>




```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm8:'00', 32);
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

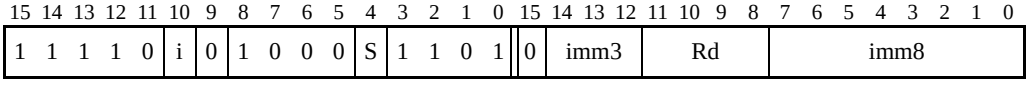
ADD<c> SP,SP,#<imm7>




```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

エンコード T3 ARMv6T2、ARMv7

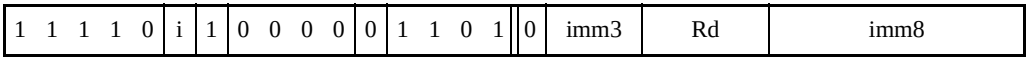
ADD{S}<c>.W <Rd>,SP,#<const>




```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
d = UInt(Rd);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 15 then UNPREDICTABLE;
```

エンコード T4 ARMv6T2、ARMv7

ADDW<c> <Rd>,SP,#<imm12>




```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

アセンブラ構文

ADD{S}<c><q> {<Rd>,} SP, #<const>	すべてのエンコードが許可される
ADDW<c><q> {<Rd>,} SP, #<const>	エンコード T4 のみが許可される

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。<Rd> を省略した場合は、SP が使用されます。

<const>　<Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲は、エンコード T1 では 4 の倍数で 0 ～ 1020、エンコード T2 では 4 の倍数で 0 ～ 508、エンコード T4 では 0 ～ 4095 の任意の値です。エンコード T3 で許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

両方の 32 ビットエンコードが使用可能な場合、エンコード T4 よりもエンコード T3 の方が適しています（T4 エンコードが必要な場合は、ADDW 構文を使用して下さい）。

UAL 以前の構文の ADD<c>S は、ADDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, imm32, '0');
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

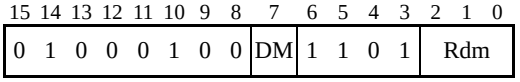
なし

A6.7.6 ADD（SP + レジスタ）

ADD（加算（SP + レジスタ））は、オプションでシフトされたレジスタの値を SP の値に加算して、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

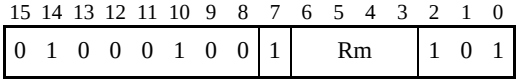
ADD<c> <Rdm>, SP, <Rdm>




```
d = UInt(DM:Rdm);  m = UInt(DM:Rdm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

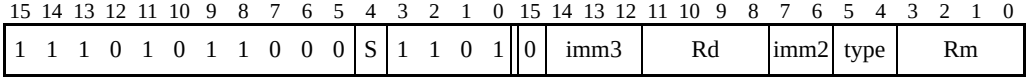
ADD<c> SP,<Rm>




```
if Rm == '1101' then SEE encoding T1;
d = 13;  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T3 ARMv6T2、ARMv7

ADD{S}<c>.W <Rd>,SP,<Rm>{,<shift>}




```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 && (shift_t != SRType_LSL || shift_n > 3) then UNPREDICTABLE;
if d == 15 || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

ADD{S}<c><q> {<Rd>,} SP, <Rm>{, <shift>}

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。<Rd> を省略した場合、SP が使用されます。

<Rm>　オプションでシフトされた、2 番目のオペランドとして使用されるレジスタを指定します。このレジスタには SP を指定できますが、次の点を考慮する必要があります。

- SP の使用は推奨されません。
- 使用できるエンコードは T1 のみで、命令として ADD SP,SP,SP だけが許可されます。

<shift>　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われず、すべてのエンコードが許可されます。<shift> を指定した場合、エンコード T3 のみが使用できます。どのようなシフトが使用可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

<Rd> が SP の場合、または省略されている場合は、<shift> には LSL #0、LSL #1、LSL #2、LSL #3 のみが使用できます。

UAL 以前の構文の ADD<c>S は、ADDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(SP, shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

例外

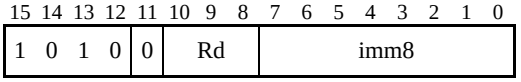
なし

A6.7.7 ADR

アドレスのレジスタへの転送（ADR）は、イミディエート値を PC の値に加算して、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

ADR<c> <Rd>,<label>




```
d = UInt(Rd);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

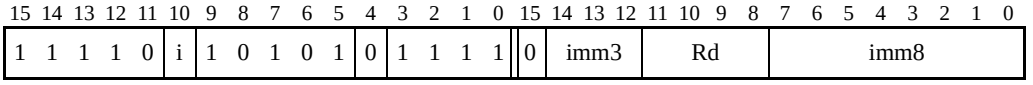
エンコード T2 ARMv6T2、ARMv7

ADR<c>.W <Rd>,<label>

<label> は現在の命令の前

SUB <Rd>,PC,#0

オフセットが 0 の特殊なケース

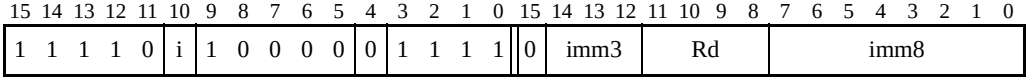



```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = FALSE;
if BadReg(d) then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

ADR<c>.W <Rd>,<label>

<label> は現在の命令の後




```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = TRUE;
if BadReg(d) then UNPREDICTABLE;
```

アセンブラ構文

ADR<c><q> <Rd>, <label>	通常の構文
ADD<c><q> <Rd>, PC, #<const>	エンコード T1 および T3 用の代替構文
SUB<c><q> <Rd>, PC, #<const>	エンコード T2 用の代替構文

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<label> アドレスが <Rd> にロードされることになる命令またはリテラルデータ項目のラベルを指定します。アセンブラは、ADR 命令の Align(PC,4) の値からこのラベルまでのオフセットに必要な値を計算します。

オフセットが正の場合、エンコード T1 および T3 が許可され、imm32 はオフセットに等しくなります。オフセットに許容される値の範囲は、エンコード T1 では 4 の倍数で 0 ～ 1020、エンコード T3 では 0 ～ 4095 の任意の値です。

オフセットが負の場合、エンコード T2 が許可され、imm32 はオフセットの負の値に等しくなります。オフセットに許容される値の範囲は -4095 ～ -1 です。

代替構文形式の場合：

<const> ADD 書式ではオフセット値を、SUB 書式ではオフセットの負の値を指定します。オフセットに許容される値の範囲は、エンコード T1 では 4 の倍数で 0 ～ 1020、エンコード T2 および T3 では 0 ～ 4095 の任意の値です。

注

可能であれば、代替構文形式は使用しないことをお勧めします。ただし、エンコード T2 ですべての immediate ビットが 0 の場合、使用できる構文は SUB<c><q> <Rd>,PC,#0 のみです。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    R[d] = result;
```

例外

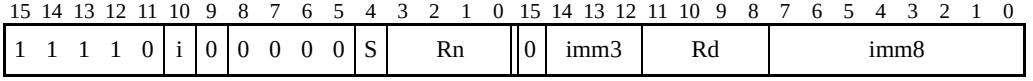
なし

A6.7.8 AND（イミディエート）

AND（論理積（イミディエート））は、レジスタの値とイミディエート値との間でビット単位の論理積を実行し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

AND{S}<c> <Rd>,<Rn>,#<const>




```
if Rd == '1111' && S == '1' then SEE TST (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

アセンブラ構文

AND{S}<c><q> {<Rd>,} <Rn>, #<const>

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

<Rn>　第一オペランドを格納しているレジスタを指定します。

<const>　<Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

UAL 以前の構文の AND<c>S は、ANDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

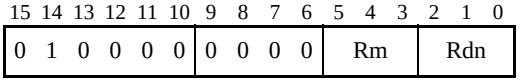
A6.7.9 AND（レジスタ）

AND（論理積（レジスタ））は、レジスタの値と、オプションとしてシフトしたレジスタの値との間でビット単位の論理積を実行し、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

ANDS <Rdn>,<Rm> IT ブロック外部

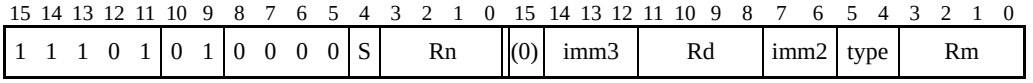
AND<c> <Rdn>,<Rm> IT ブロック内部




```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

AND{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}




```
if Rd == '1111' && S == '1' then SEE TST (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

AND{S}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

<Rn>　第一オペランドを格納しているレジスタを指定します。

<Rm>　オプションでシフトされて、2 番目のオペランドとして使用されるレジスタを指定します。

<shift>　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われず、両方のエンコードが許容されます。<shift> を指定した場合は、エンコード T2 のみが許容されます。どのシフトが許容されるか、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

特殊なケースとして、AND<c> <Rd>,<Rn>,<Rd> と記述し、<Rd> および <Rn> がいずれも R0 ～ R7 の場合は、AND<c> <Rd>,<Rn> と記述されたものとして、エンコード T2 を使用してアセンブルされます。これを避けるには、.W 修飾子を使用します。

UAL 以前の構文の AND<c>S は、ANDS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

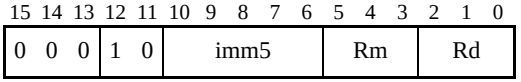
A6.7.10 ASR（イミディエート）

ASR（算術右シフト（イミディエート））は、レジスタの値のビットをイミディエート値で指定されたビット数だけ右にシフト（空いたビットには、符号ビットがコピーされます）して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

ASRS <Rd>,<Rm>,#<imm5> IT ブロック外部

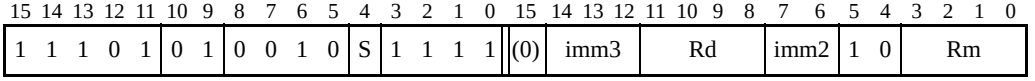
ASR<c> <Rd>,<Rm>,#<imm5> IT ブロック内部




```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('10', imm5);
```

エンコード T2 ARMv6T2、ARMv7

ASR{S}<c>.W <Rd>,<Rm>,#<imm5>




```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('10', imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
ASR{S}<c><q>  <Rd>, <Rm>, #<imm5>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rm>　　第一オペランドを格納しているレジスタを指定します。

<imm5>　　シフトするビット数を 1 ～ 32 の範囲で指定します。詳細については、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_ASR, shift_n, APSR.C);
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

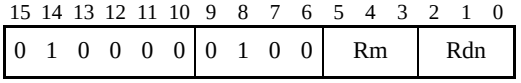
A6.7.11 ASR（レジスタ）

ASR（算術右シフト（レジスタ））は、レジスタの値を可変のビット数だけ右にシフト（空いたビットには、符号ビットがコピーされます）し、結果をデスティネーションレジスタに書き込みます。可変のビット数は、レジスタの下位バイトから読み出されます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

ASRS <Rdn>,<Rm> IT ブロック外部

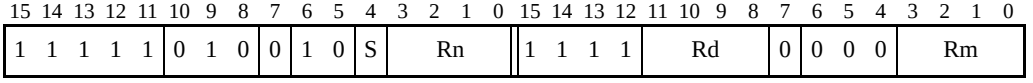
ASR<c> <Rdn>,<Rm> IT ブロック内部




```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

エンコード T2 ARMv6T2、ARMv7

ASR{S}<c>.W <Rd>,<Rn>,<Rm>




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```


アセンブラ構文

ASR{S}<c><q> <Rd>, <Rn>, <Rm>

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。

<Rn>　第一オペランドを格納しているレジスタを指定します。

<Rm>　下位バイトにシフトのビット数を含むレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_ASR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

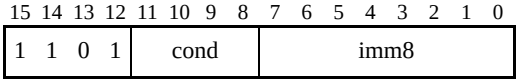
A6.7.12 B

B（分岐）は、ターゲットアドレスへの分岐を発生させます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

B<c> <label>

IT ブロック内部では許可されない

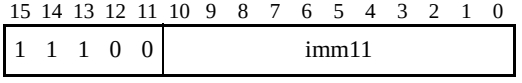



```
if cond == '1110' then UNDEFINED;
if cond == '1111' then SEE SVC;
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

B<c> <label>

IT ブロック外部または IT ブロック内部の最後

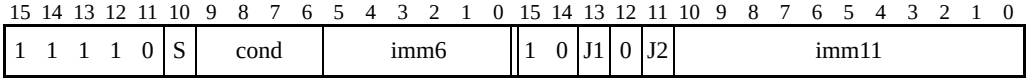



```
imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

B<c>.W <label>

IT ブロック内部では許可されない

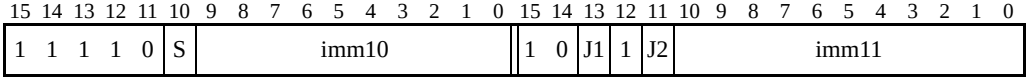



```
if cond<3:1> == '111' then SEE "Related encodings";
imm32 = SignExtend(S:J2:J1:imm6:imm11:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

エンコード T4 ARMv6T2、ARMv7

B<c>.W <label>

IT ブロック外部または IT ブロック内部の最後




```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

アセンブラ構文

B<c><q> <label>

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

注

エンコード T1 および T3 はそれ自体が条件付きで、IT 命令を使用して条件付きにする必要はありません。

エンコード T1 および T3 では、<c> に AL を指定できず、省略もできません。条件の 4 ビットエンコードは、先行する IT 命令内ではなく、この命令に含まれるため、この命令は IT ブロック内で使用できません。結果として、エンコード T1 および T2 の両方、またはエンコード T3 および T4 の両方をアセンブラで同時に使用することはできません。

<label> 分岐先の命令のラベルを指定します。アセンブラは、B 命令の PC の値からこのラベルまでのオフセットに必要な値を計算し、imm32 をそのオフセットに設定するエンコードを選択します。

許容されるオフセットの範囲は、エンコード T1 では -256 ～ 254、エンコード T2 では -2048 ～ 2046、エンコード T3 では -1048576 ～ 1048574、エンコード T4 では -16777216 ～ 16777214 で、どのエンコードでも偶数のみが使用できます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32);
```

例外

なし

関連エンコード

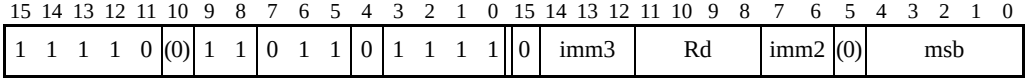
エンコード T3 の cond フィールドが "1110" または "1111" の場合、別の命令がエンコードされます。どの命令かを判断するには、P.A5-18「*分岐およびその他の制御*」を参照して下さい。

A6.7.13 BFC

BFC（ビットフィールド クリア）は、レジスタ内の任意の位置にある、隣接する任意の数のビットをクリアします。レジスタ内の他のビットには影響を与えません。

エンコード T1 ARMv6T2、ARMv7

BFC<c> <Rd>,#<lsb>,#<width>




```
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if BadReg(d) then UNPREDICTABLE;
```

アセンブラ構文

```
BFC<c><q>  <Rd>, #<lsb>, #<width>
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<lsb> クリアする最下位ビットを、0 ～ 31 の範囲で指定します。これによって、lsbit に必要な値が決定されます。

<width> クリアするビット数を、1 ～ 32-<lsb> の範囲で指定します。msbit に必要な値は <lsb>+<width>-1 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = Replicate('0', msbit-lsbit+1);
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

例外

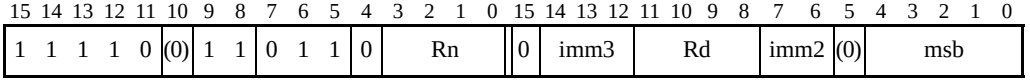
なし

A6.7.14 BFI

BFI（ビットフィールド挿入）は、ソースレジスタから任意の数の低位ビットを、デスティネーションレジスタの任意の位置にある同じ数の隣接するビットにコピーします。

エンコードT1 ARMv6T2、ARMv7

BFI<c> <Rd>,<Rn>,#<lsb>,#<width>




```
if Rn == '1111' then SEE BFC;
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if BadReg(d) || n == 13 then UNPREDICTABLE;
```

アセンブラ構文

BFI<c><q> <Rd>, <Rn>, #<lsb>, #<width>

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<Rn> ソースレジスタを指定します。

<lsb> 最下位デスティネーションビットを指定します。

<width> コピーするビット数を指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = R[n]<(msbit-lsbit):0>;
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

例外

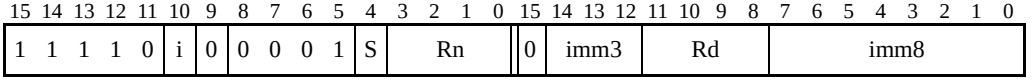
なし

A6.7.15 BIC（イミディエート）

BIC（ビットクリア（イミディエート））は、レジスタの値と、イミディエート値の補数とのビット単位の論理積を実行し、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

BIC{S}<c> <Rd>,<Rn>,#<const>




```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

アセンブラ構文

BIC{S}<c><q> {<Rd>,} <Rn>, #<const>

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。<Rd> を省略した場合は、このレジスタが <Rn> になります。

<Rn>　オペランドを含むレジスタを指定します。

<const>　<Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

UAL 以前の構文の BIC<c>S は、BICS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND NOT(imm32);
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

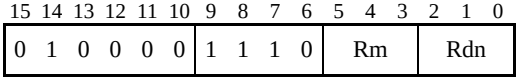
A6.7.16 BIC（レジスタ）

BIC（ビットクリア（レジスタ））は、レジスタの値と、オプションでシフトされたレジスタの値の補数とでビット単位の論理積を実行し、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

BICS <Rdn>,<Rm> IT ブロック外部

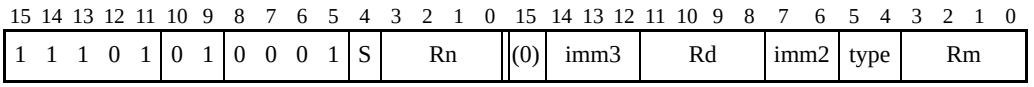
BIC<c> <Rdn>,<Rm> IT ブロック内部




```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

BIC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

BIC{S}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

<Rn>　第一オペランドを格納しているレジスタを指定します。

<Rm>　オプションでシフトされて、2 番目のオペランドとして使用されるレジスタを指定します。

<shift>　<Rm> から読み出された値に適用するシフトを指定します。<shift> を省略した場合、シフトは行われず、両方のエンコードが許可されます。<shift> を指定した場合は、エンコード T2 のみが使用できます。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

UAL 以前の構文の BIC<c>S は、BICS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND NOT(shifted);
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

A6.7.17 BKPT

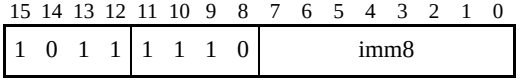
BKPT（ブレークポイント）は、デバッグサポートの構成に応じて、DebugMonitor 例外またはデバッグホールトを発生させます。

注

BKPT は無条件命令で、IT 命令ブロックの内部の内部でも外部でも無条件で動作します。

エンコード T1 ARMv5 またはそれ以降のすべてのバージョンの Thumb ISA

BKPT #<imm8>




```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly only and is ignored by hardware.
```

アセンブラ構文

BKPT<q> #<imm8>

各項目の説明については以下を参照して下さい。

<q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<imm8> 命令に格納される 8 ビット値を指定します。この値は、ARM ハードウェアでは無視されますが、デバッガでブレークポイントに関する追加情報の保持に使用できます。

動作

```
EncodingSpecificOperations();
BKPTInstrDebugEvent();
```

例外

デバッグモニタ

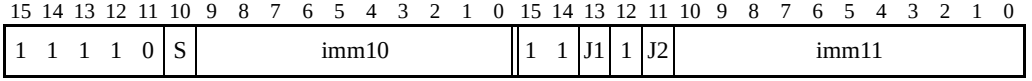
A6.7.18 BL

リンク付き分岐 (BL) （イミディエート）は、PC 相対アドレスでサブルーチンを呼び出します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

BL<c> <label>

IT ブロック外部または IT ブロック内部の最後




```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
toARM = FALSE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

アセンブラ構文

```
BL<c><q>  <label>
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<label> 分岐先の命令のラベルを指定します。
アセンブラは、BL 命令の PC の値からこのラベルまでのオフセットに必要な値を計算して、imm32 をそのオフセットに設定するエンコードを選択します。オフセットに許容される値の範囲は、-16777216 ～ 16777214 までの偶数です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    next_instr_addr = PC;
    LR = next_instr_addr<31:1> : '1';
    BranchWritePC(PC + imm32);
```

例外

なし

注

Thumb-2 以前では、エンコード T1 および T2 の J1 および J2 はいずれも 1 で、結果的に分岐範囲が制限されます。また、Thumb-2 以前では、この命令を 2 つの異なる 16 ビット命令として実行することができます。最初の命令の instr1 が LR を PC + SignExtend(instr1<10:0>:'000000000000', 32) に設定し、2 番目の命令が動作を完了させます。Thumb-2 では、BL 命令を 2 つの 16 ビット命令に分割することができなくなっています。

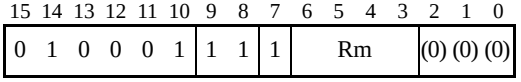
A6.7.19 BLX（レジスタ）

BLX（リンク付き分岐と状態遷移）は、レジスタで指定されたアドレスにある、指定された命令セットのサブルーチンを呼び出します。ARMv7-M では、Thumb 命令セットのみがサポートされています。命令実行状態を変更しようとすると、例外が発生します。

エンコード T1　ARMv5 またはそれ以降のすべてのバージョンの Thumb ISA

BLX<c> <Rm>

IT ブロック外部または IT ブロック内部の最後



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	1	Rm				(0)	(0)	(0)

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

アセンブラ構文

```
BLX<c><q>  <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rm>　分岐先アドレスと命令セット選択ビットを含むレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    next_instr_addr = PC - 2;
    LR = next_instr_addr<31:1> : '1';
    BXWritePC(R[m]);
```

例外

UsageFault

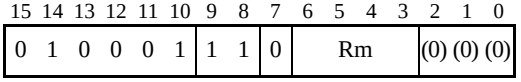
A6.7.20 BX

BX（分岐と状態遷移）は、レジスタで指定されたアドレスにある、指定された命令セットへの分岐を発生させます。ARMv7-M では、Thumb 命令セットのみがサポートされています。命令実行状態を変更しようとすると、例外が発生します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

BX<c> <Rm>

IT ブロック外部または IT ブロック内部の最後



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0	Rm				(0)	(0)	(0)

```
m = UInt(Rm);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

アセンブラ構文

```
BX<c><q>  <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rm> 分岐先アドレスと命令セット選択ビットを含むレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m]);
```

例外

UsageFault

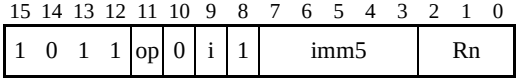
A6.7.21 CBNZ, CBZ

CBNZ（比較して非 0 で分岐）および CBZ（比較して 0 で分岐）は、レジスタ内の値と 0 を比較して、条件付きで定数値分の前方分岐を実行します。条件フラグには影響を与えません。

エンコード T1 ARMv6T2、ARMv7

CB{N}Z <Rn>,<label>

IT ブロック内部では許可されない




```
n = UInt(Rn);  imm32 = ZeroExtend(i:imm5:'0', 32);  nonzero = (op == '1');
if InITBlock() then UNPREDICTABLE;
```

アセンブラ構文

```
CB{N}Z<q>  <Rn>, <label>
```

各項目の説明については以下を参照して下さい。

<q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　　最初のオペランドレジスタ

<label>　　分岐先の命令のラベル。アセンブラは、CB{N}Z 命令の PC 値からこのラベルまでのオフセットに必要な値を計算して、imm32 をそのオフセットに設定するエンコードを選択します。オフセットに許容される値の範囲は、0 ～ 126 までの偶数です。

動作

```
EncodingSpecificOperations();
if nonzero ^ IsZero(R[n]) then
    BranchWritePC(PC + imm32);
```

例外

なし

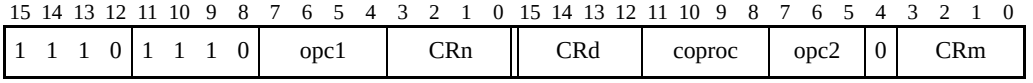
A6.7.22 CDP, CDP2

CDP、CDP2（コプロセッサデータ処理）は、コプロセッサに対して、ARM レジスタやメモリと独立して動作を実行するように指示します。

コプロセッサが命令を実行できない場合、UsageFault 例外が発生します。

エンコード T1 すべての Thumb-2 ISA

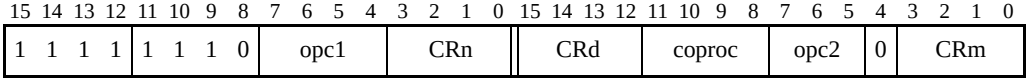
CDP<c> <coproc>,<opc1>,<CRd>,<CRn>,<CRm>,<opc2>




```
cp = UInt(coproc);
```

エンコード T2 すべての Thumb-2 ISA

CDP2<c> <coproc>,<opc1>,<CRd>,<CRn>,<CRm>,<opc2>




```
cp = UInt(coproc);
```

アセンブラ構文

CDP{2}<c><q> <coproc>, #<opc1>, <CRd>, <CRn>, <CRm> {,#<opc2>}

各項目の説明については以下を参照して下さい。

2 指定した場合、エンコードの opc0 == 1 形式が選択されます。省略した場合、opc0 == 0 形式が選択されます。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<coproc> コプロセッサの名前を指定します。対応するコプロセッサ番号が命令の cp_num フィールドに配置されます。標準の汎用コプロセッサ名は、p0、p1、...、p15 です。

<opc1> コプロセッサ固有のオペコードで、範囲は 0 〜 15 です。

<CRd> 命令のデスティネーションコプロセッサ レジスタを指定します。

<CRn> 第一オペランドを格納しているコプロセッサレジスタを指定します。

<CRm> 2 番目のオペランドを格納しているコプロセッサレジスタを指定します。

<opc2> コプロセッサ固有のオペコードで、範囲は 0 〜 7 です。省略した場合は、<opc2> が 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_InternalOperation(cp, ThisInstr());
```

例外

UsageFault

注

コプロセッサのフィールド

アーキテクチャ上は、命令のビット <31:24>、ビット <11:8>、およびビット <4> だけが定義されています。その他のフィールドは推奨事項です。

A6.7.23 CLREX

CLREX（排他クリア）は、特定のアドレスに対して排他アクセスを要求した実行中のプロセッサのローカルレコードをクリアします。

エンコード T1 ARMv6K、ARMv7

CLREX<c>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	0	1	0	(1)	(1)	(1)	(1)

```
// No additional decoding required
```

アセンブラ構文

CLREX<c><q>

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
```

例外

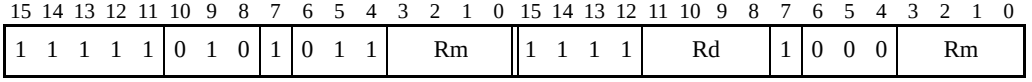
なし

A6.7.24 CLZ

CLZ（先行ゼロカウント）は、値の中で、最初に存在する 1 のバイナリビットより前にある 0 のバイナリビットの数を返します。

エンコード T1　　ARMv6T2、ARMv7

CLZ<c> <Rd>,<Rm>




```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
CLZ<c><q>  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rm>　　オペランドを格納しているレジスタを指定します。エンコード T1 では、RmRm フィールドおよび Rm2 フィールドの両方で、その数を 2 回エンコードする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = CountLeadingZeroBits(R[m]);
    R[d] = result<31:0>;
```

例外

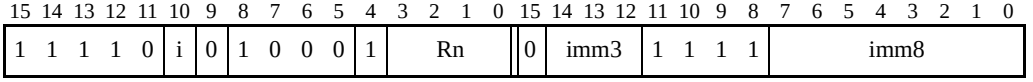
なし

A6.7.25 CMN（イミディエート）

CMN（比較否定（イミディエート））は、レジスタの値とイミディエート値を加算します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv6T2、ARMv7

CMN<c> <Rn>,#<const>




```
n = UInt(Rn);  imm32 = ThumbExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```


アセンブラ構文

```
CMN<c><q>  <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　　オペランドを含むレジスタを指定します。このレジスタには SP を指定できます。

<const>　　<Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

例外

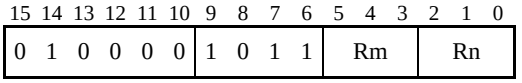
なし

A6.7.26 CMN（レジスタ）

CMN（比較否定（レジスタ））は、レジスタの値と、オプションとしてシフトしたレジスタの値を加算します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

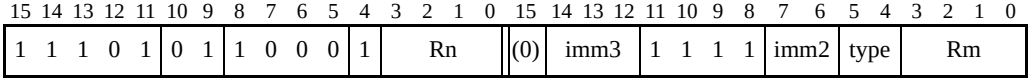
CMN<c> <Rn>,<Rm>




```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

CMN<c>.W <Rn>,<Rm>{,<shift>}




```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
CMN<c><q>  <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　第一オペランドを格納しているレジスタを指定します。SP を指定できます。

<Rm>　オプションでシフトされる、2 番目のオペランドとして使用されるレジスタを指定します。

<shift>　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われず、両方のエンコードが許可されます。<shift> を指定した場合、エンコード T2 のみが許可されます。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, ‘0’);
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

例外

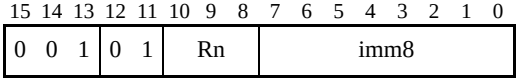
なし

A6.7.27 CMP（イミディエート）

CMP（比較（イミディエート））は、レジスタの値からイミディエート値を減算します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

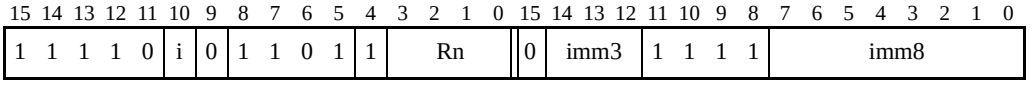
CMP<c> <Rn>,#<imm8>




```
n = UInt(Rdn);  imm32 = ZeroExtend(imm8, 32);
```

エンコード T2 ARMv6T2、ARMv7

CMP<c>.W <Rn>,#<const>




```
n = UInt(Rn);  imm32 = ThumbExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

アセンブラ構文

CMP<c><q> <Rn>, #<const>

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn> オペランドを含むレジスタを指定します。SP を指定できます。

<const> <Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲は、エンコード T1 では 0 ～ 255 です。エンコード T2 で許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

例外

なし

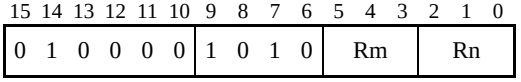
A6.7.28 CMP（レジスタ）

CMP（比較（レジスタ））は、レジスタの値から、オプションとしてシフトしたレジスタの値を減算します。また、結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1　　ARMv4T、ARMv5T*、ARMv6*、ARMv7

CMP<c> <Rn>,<Rm>

<Rn> および <Rm> の両方が R0 ～ R7

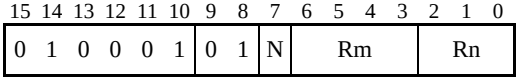



```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2　　ARMv4T、ARMv5T*、ARMv6*、ARMv7

CMP<c> <Rn>,<Rm>

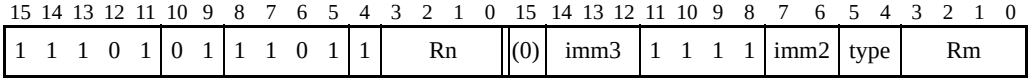
<Rn> および <Rm> の両方が R0 ～ R7 ではない




```
n = UInt(N:Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
if n < 8 && m < 8 then UNPREDICTABLE;
if n == 15 || m == 15 then UNPREDICTABLE;
```

エンコード T3　　ARMv6T2、ARMv7

CMP<c>.W <Rn>, <Rm> {,<shift>}




```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
CMP<c><q>  <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　第一オペランドを格納しているレジスタを指定します。このレジスタには SP を指定できます。

<Rm>　オプションでシフトされる、2 番目のオペランドとして使用されるレジスタを指定します。SP を使用することができますが、SP の使用は推奨されません。

<shift>　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われず、すべてのエンコードが許可されます。shift を指定した場合、エンコード T3 のみが許可されます。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

例外

なし

A6.7.29 CPS

CPS（プロセッサ状態変更）は、PRIMASK および FAULTMASK 専用レジスタの値を変更します。

エンコード T1 ARMv6*、ARMv7 IT ブロック内部では許可されない

CPS<effect> <iflags>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	(0)	I	F

注

CPS は、ARMv7-M 固有の動作を行うシステムレベル命令です。完全な命令定義については、P.B3-2「*CPS*」を参照して下さい。

A6.7.30 CPY

コピー (CPY) は、UAL 以前の MOV（レジスタ）と同義語です。

アセンブラ構文

```
CPY  <Rd>, <Rn>
```

これは、次の命令と等価です。

MOV <Rd>, <Rn>

例外

なし

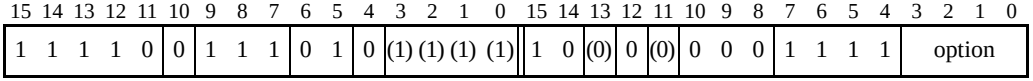
A6.7.31 DBG

DBG（デバッグヒント）は、デバッグトレース サポートと、関連するデバッグシステムに対して、ヒントを提供します。この命令の使用方法については、デバッグアーキテクチャの説明書を参照して下さい。

これは、NOP 互換ヒントです。全般的なヒントの動作については、P.A6-16「*NOP 互換ヒント*」を参照して下さい。

エンコード T1 ARMv7（ARMv6T2 では NOP として動作する）

```
DBG<c> #<option>
```




```
// Any decoding of ‘option’ is specified by the debug system
```

アセンブラ構文

```
DBG<c><q>  #<option>
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<option> ヒントに関するその他の情報を提供します。範囲は 0 ～ 15 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Debug(option);
```

例外

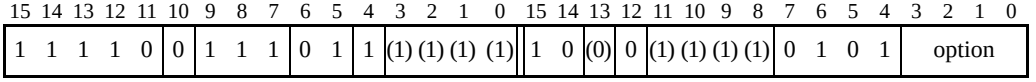
なし

A6.7.32 DMB

DMB（データメモリ バリア）は、メモリバリアとして機能します。この命令によって、プログラムの順序で DMB 命令の前に出現するすべての明示的なメモリアクセスが、プログラムの順序で DMB 命令の後に出現する任意の明示的なメモリアクセスの前に観測されることが保証されます。プロセッサ上で実行される他の命令の順序には影響を与えません。

エンコード T1 ARMv7

DMB<c> #<option>




```
// No additional decoding required
```


アセンブラ構文

```
DMB<c><q>  {<opt>}
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<opt>　　オプションで、DMB 動作に対する制限を指定します。

SY　　DMB 動作により、すべてのアクセスの順序を保証します。option == '1111' としてエンコードされます。省略可能。

オプションの他のエンコードはすべて予約されています。これらに対応する命令はシステムの (SY) DMB 動作として機能しますが、ソフトウェアではこの動作を前提としないようにする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    DataMemoryBarrier(option);
```

例外

なし

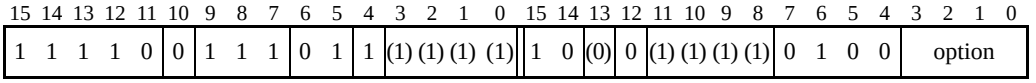
A6.7.33 DSB

DSB（データ同期バリア）は、メモリバリアの特殊な形式として機能します。プログラムの順序でこの命令より後の命令は、この命令が完了するまで実行できません。この命令は、次の場合に完了します。

- この命令より前のすべての明示的なメモリアクセスが完了したとき
- この命令より前のすべてのキャッシュ、分岐予測器、および TLB の保守操作が完了したとき

エンコード T1 ARMv7

DMB<c> #<option>




```
// No additional decoding required
```

アセンブラ構文

```
DSB<c><q>  {<opt>}
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<opt>　オプションで、DSB 動作に対する制限を指定します。

SY　DSB動作により、すべてのアクセスの完了を保証します。option == '1111' としてエンコードされます。省略可能

オプションの他のエンコードはすべて予約されています。これらに対応する命令はシステムの (SY) DSB 動作として機能しますが、ソフトウェアではこの動作を前提としないようにする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    DataSynchronizationBarrier(option);
```

例外

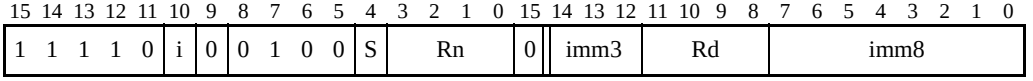
なし

A6.7.34 EOR（イミディエート）

EOR（排他的論理和（イミディエート））は、レジスタの値とイミディエート値との間でビット単位の排他的論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

EOR{S}<c> <Rd>,<Rn>,#<const>




```
if Rd == '1111' && S == '1' then SEE TEQ (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

アセンブラ構文

```
EOR{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合は、<Rn> と同じレジスタが使用されます。

<Rn>　　オペランドを含むレジスタを指定します。

<const>　　<Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

UAL 以前の構文の EOR<c>S は、EORS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

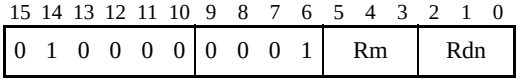
A6.7.35 EOR（レジスタ）

EOR（排他的論理和（レジスタ））は、レジスタの値と、オプションとしてシフトしたレジスタの値との間でビット単位の排他的論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

EORS <Rdn>,<Rm> IT ブロック外部

EOR<c> <Rdn>,<Rm> IT ブロック内部

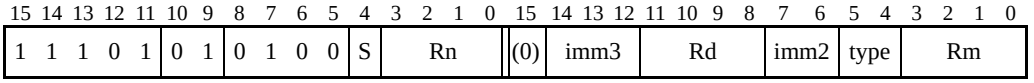


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

EOR{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	1	0	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
if Rd == '1111' && S == '1' then SEE TEQ (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
EOR{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> と同じレジスタが使用されます。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　オプションでシフトされる、2 番目のオペランドとして使用されるレジスタを指定します。

<shift>　　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われず、両方のエンコードが許可されます。<shift> を指定した場合、エンコード T2 のみが許可されます。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

特殊なケースとして、EOR<c> <Rd>,<Rn>,<Rd> と記述し、<Rd> および <Rn> の両方が R0 ～ R7 の範囲内の場合、EOR<c> <Rd>,<Rn> と記述されたものとして、エンコード T2 を使用してアセンブルされます。これを避けるには、.W 修飾子を使用します。

UAL 以前の構文の EOR<c>S は、EORS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

A6.7.36 ISB

ISB（命令同期バリア）は、プロセッサ内のパイプラインをフラッシュして、ISB 命令よりも後にあるすべての命令は、この命令の完了後にキャッシュまたはメモリからフェッチされるようにします。これによって、ISB 命令の前に実行された、ASID の変更、TLB 保守操作の完了、分岐予測器保守操作の完了などのコンテキスト変更動作、および CP15 レジスタに対するすべての変更の影響が、ISB 命令の後にフェッチされた命令から可視であることが保証されます。

さらに、ISB 命令によって、この命令の後にプログラムの順で出現するすべての分岐が、ISB 命令の後で可視のコンテキストで分岐検出ロジックに書き込まれることが保証されます。これは、命令ストリームが正しく実行されることを保証するために必要となる条件です。

エンコード T1 ARMv7

ISB<c> #<option>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	0	option			

```
// No additional decoding required
```

アセンブラ構文

```
ISB<c><q>  {<opt>}
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<opt>　ISB 動作に対するオプションの制限を指定します。許容される値は次のとおりです。

SY　完全なシステム ISB 動作を指定し、option == '1111' としてエンコードされます。省略可能

他のオプションのエンコードはすべて予約されています。これらに対応する命令は完全なシステム ISB 動作として実行されますが、ソフトウェアではこの動作を前提にしないようにする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier(option);
```

例外

なし

A6.7.37 IT

IT（If Then）は、以降の最大 4 つの命令（IT ブロック）を条件付きにします。IT ブロック内部の命令の条件は、すべてを同じにすることも、一部を反転させることもできます。

IT は、条件コードフラグに影響を与えません。IT ブロック内部の任意の命令への分岐は、例外復帰で実行される以外は許可されません。

CMP、CMN、TST を除く IT ブロック内部の 16 ビット命令では、条件コードフラグがセットされません。AL 条件を指定すると、条件付き実行を行わず、動作だけをこのように変更することができます。

エンコード T1 ARMv6T2、ARMv7

IT{x{y{z}}} <firstcond> IT ブロック内部では許可されない

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	firstcond				mask			

```
if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' then UNPREDICTABLE;
if firstcond == '1110' && BitCount(mask) != 1 then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

アセンブラ構文

```
IT{x{y{z}}}<q>  <firstcond>
```

各項目の説明については以下を参照して下さい。

<x> IT ブロック内部の 2 番目の命令に対する条件を指定します。

<y> IT ブロック内部の 3 番目の命令に対する条件を指定します。

<z> IT ブロック内部の 4 番目の命令に対する条件を指定します。

<q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<firstcond> IT ブロック内部の最初の命令に対する条件を指定します。

<x>、<y>、<z> には次のいずれかを指定できます。

T Then。命令に付加される条件は <firstcond> です。

E Else。命令に付加される条件は <firstcond> の逆です。条件コードは、最下位ビットが反転していること以外、<firstcond> と同じです。<firstcond> が AL の場合は、E を指定できません。

表 A6-3 に示すように、<x>、<y>、<z> の値によって mask フィールドの値が決定されます。

表 A6-3 mask [a] フィールドの値

<x>	<y>	<z>	mask[3]	mask[2]	mask[1]	mask[0]
省略	省略	省略	1	0	0	0
T	省略	省略	firstcond[0]	1	0	0
E	省略	省略	NOT firstcond[0]	1	0	0
T	T	省略	firstcond[0]	firstcond[0]	1	0
E	T	省略	NOT firstcond[0]	firstcond[0]	1	0
T	E	省略	firstcond[0]	NOT firstcond[0]	1	0
E	E	省略	NOT firstcond[0]	NOT firstcond[0]	1	0
T	T	T	firstcond[0]	firstcond[0]	firstcond[0]	1
E	T	T	NOT firstcond[0]	firstcond[0]	firstcond[0]	1
T	E	T	firstcond[0]	NOT firstcond[0]	firstcond[0]	1
E	E	T	NOT firstcond[0]	NOT firstcond[0]	firstcond[0]	1
T	T	E	firstcond[0]	firstcond[0]	NOT firstcond[0]	1
E	T	E	NOT firstcond[0]	firstcond[0]	NOT firstcond[0]	1
T	E	E	firstcond[0]	NOT firstcond[0]	NOT firstcond[0]	1
E	E	E	NOT firstcond[0]	NOT firstcond[0]	NOT firstcond[0]	1

a. マスク内では常に 1 のビットが 1 つ以上存在する必要があることに注意して下さい。

P.A6-10「*ITSTATE*」も参照して下さい。

動作

```
EncodingSpecificOperations();
ITSTATE.IT<7:0> = firstcond:mask;
```

例外

なし

関連エンコード

エンコード T1 の mask フィールドが "0000" の場合は、別の命令がエンコードされます。どの命令かを判断するには、P.A5-11「*If-Then とヒント*」を参照して下さい。

A6.7.38 LDC、LDC2（イミディエート）

LDC、LDC2（コプロセッサロード（イミディエート））は、連続するメモリアドレスからコプロセッサにメモリデータをロードします。コプロセッサが命令を実行できない場合は、UsageFault 例外が発生します。

これは、汎用のコプロセッサ命令です。一部のフィールドの機能はアーキテクチャで定義されていないため、コプロセッサ命令セットの設計者が自由に使用することができます。これらのフィールドは、D ビット、CRd フィールド、および imm8 フィールド（インデクスなしアドレシングモードのみ）です。

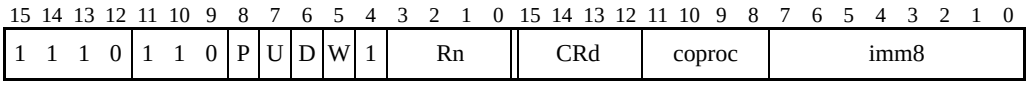
エンコード T1 すべての ARM および Thumb-2 ISA

LDC{L}<c> <coproc>,<CRd>,[<Rn>{,#+/-<imm>}]

LDC{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm>]!

LDC{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm>

LDC{L}<c> <coproc>,<CRd>,[<Rn>],<option>




```
if Rn == '1111' then SEE LDC (literal);
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

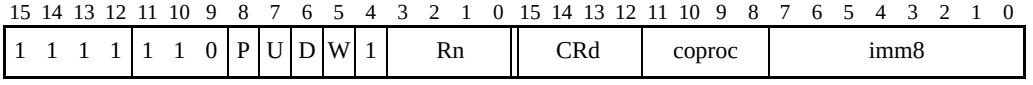
エンコード T2 すべての Thumb-2 ISA、ARMv5 またはそれ以降のすべての ARM ISA

LDC2{L}<c> <coproc>,<CRd>,[<Rn>{,#+/-<imm>}]

LDC2{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm>]!

LDC2{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm>

LDC2{L}<c> <coproc>,<CRd>,[<Rn>],<option>




```
if Rn == '1111' then SEE LDC (literal);
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

アセンブラ構文

LDC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>{,#+/-<imm>}]	オフセット：P = 1、W = 0
LDC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>,#+/-<imm>]!	プリインデクス：P = 1、W = 1
LDC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],#+/-<imm>	ポストインデクス：P = 0、W = 1
LDC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],<option>	インデクスなし：P = 0、W = 0、U = 1

各項目の説明については以下を参照して下さい。

L　指定した場合、エンコードの D == 1 の形式が選択されます。省略した場合、D == 0 の形式が選択されます。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<coproc>　コプロセッサの名前。標準の汎用コプロセッサ名は、p0、p1、...、p15 です。

<CRd>　コプロセッサのデスティネーションレジスタ

<Rn>　ベースレジスタ。SP または PC が指定できます。

+/-　イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm>　アドレス生成のために <Rn> の値に適用されるイミディエートオフセット。許容される値の範囲は、4 の倍数で 0 ～ 1020 です。オフセットアドレシング構文では <imm> を省略でき、この場合 +0 を意味します。

<option>　コプロセッサに対する追加の命令オプション。{ } で囲んだ 0 ～ 255 の整数。imm8 にエンコードされます。

UAL 以前の構文の LDC<c>L は、LDCL<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        repeat
            Coproc_SendLoadedWord(MemA[address,4], cp, ThisInstr());  address = address + 4;
        until Coproc_DoneLoading(cp, ThisInstr());
        if wback then R[n] = offset_addr;
```

例外

UsageFault、MemManage、BusFault

A6.7.39 LDC、LDC2（リテラル）

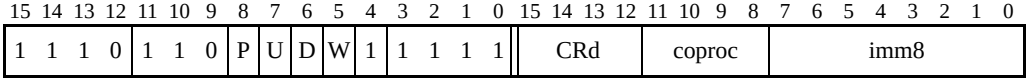
LDC、LDC2（コプロセッサロード（リテラル））は、連続するメモリアドレスからコプロセッサにメモリデータをロードします。コプロセッサが命令を実行できない場合は、UsageFault 例外が発生します。

これは、汎用のコプロセッサ命令です。D ビットと CRd フィールドの機能はアーキテクチャで定義されていないため、コプロセッサ命令セットの設計者が自由に使用することができます。

エンコード T1　　すべての ARM および Thumb-2 ISA

LDC{L}<c> <coproc>,<CRd>,label

LDC{L}<c> <coproc>,<CRd>,[PC,#-0]　　特殊なケース LDC{L}<c> <coproc>,<CRd>,[PC],<option>

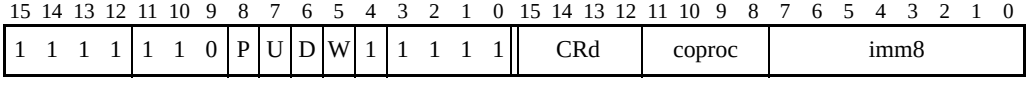



```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
index = (P == '1');     // Always TRUE in the Thumb instruction set
add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || P == '0' then UNPREDICTABLE;
```

エンコード T2　　すべての Thumb-2 ISA、ARMv5 およびそれ以降のすべての ARM ISA

LDC2{L}<c> <coproc>,<CRd>,label

LDC2{L}<c> <coproc>,<CRd>,[PC,#-0]　　特殊なケース LDC{L}<c> <coproc>,<CRd>,[PC],<option>




```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
index = (P == '1');     // Always TRUE in the Thumb instruction set
add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || P == '0' then UNPREDICTABLE;
```

アセンブラ構文

LDC{2}{L}<c><q> <coproc>,<CRd>,label　　P = 1、W = 0 の通常の形式

LDC{2}{L}<c><q> <coproc>,<CRd>,[PC,#-0]　　P = 1、W = 0 の代替形式

各項目の説明については以下を参照して下さい。

L　　指定した場合、エンコードの D == 1 の形式が選択されます。省略した場合、D == 0 の形式が選択されます。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<coproc>　　コプロセッサの名前。標準の汎用コプロセッサ名は、p0、p1、...、p15 です。

<CRd>　　コプロセッサのデスティネーションレジスタ

<label> <Rt> にロードされるリテラルデータ項目のラベル。アセンブラは、この命令の Align(PC,4) の値からこのラベルまでのオフセットに必要な値を計算します。
オフセットが正の場合、imm32 はオフセットと等しく、add == TRUE になります。
オフセットが負の場合、imm32 は負のオフセットと等しく、add == FALSE になります。

代替構文形式では、次の項目が使用されます。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために <Rn> の値に適用されるイミディエートオフセット。許容される値は、4 の倍数で 0 ～ 1020 です。<imm> は省略可能で、この場合は +0 のオフセットを意味します。

注

可能であれば、代替構文形式は使用しないことをお勧めします。ただし、U ビットおよびすべてのイミディエートビットが 0 の場合、使用できる構文は LDR<c><q> <Rt>, [PC,#-0] のみです。

UAL 以前の構文の LDC<c>L は、LDCL<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        offset_addr = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
        address = if index then offset_addr else Align(PC,4);
        repeat
            Coproc_SendLoadedWord(MemA[address,4], cp, ThisInstr());  address = address + 4;
        until Coproc_DoneLoading(cp, ThisInstr());
```

例外

UsageFault、MemManage、BusFault

A6.7.40 LDM / LDMIA / LDMFD

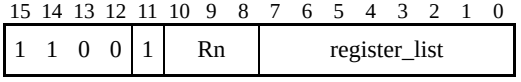
LDM、LDMIA、LDMFD（ポストインクリメント複数ロード）は、ベースレジスタからのアドレスを使用して、連続するメモリ位置から複数のレジスタをロードします。連続するメモリ位置はこのアドレスから開始します。オプションとして、ロードされる最後のメモリ位置の次のアドレスをベースアドレスに書き戻すことができます。

ロードされるレジスタには、PC を含めることができます。この場合、PC にロードされるワードは、アドレスまたは例外の戻り値として扱われ、分岐が発生します。ビット <0> は、Thumb 状態実行への分岐に関する ARM アーキテクチャのインターワーキングルールに準拠するために、1 に設定する必要があります。ビット <0> が 0 の場合は、UsageFault 例外が発生します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

LDM<c> <Rn>!,<registers> <Rn> は <registers> に含まれない

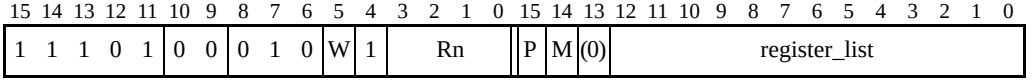
LDM<c> <Rn>,<registers> <Rn> は <registers> に含まれる




```
n = UInt(Rn);  registers = '00000000':register_list;  wback = (registers<n> == '0');
if BitCount(registers) < 1 then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

LDM<c>.W <Rn>{!},<registers>




```
if W == '1' && Rn == '1101' then SEE POP;
n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```


アセンブラ構文

```
LDM{IA|FD}<c><q>  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　ベースレジスタを指定します。このレジスタには SP を指定できます。SP を指定し、同時に ! を指定した場合、P.A6-185「*POP*」で説明されているように扱われます。

!　命令によって、修飾された値が <Rn> に書き戻されます。! を省略した場合は、命令によって <Rn> は変更されません（ただし、<Rn> が <registers> に含まれる場合は、値がロードされたときにこのレジスタが変更されます）。

<registers>　1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストで、LDM 命令によってロードされるレジスタのセットを指定します。最下位のメモリアドレスに置かれているデータは最も小さい番号のレジスタにロードされ、以下順に、最上位のメモリアドレスのデータは最も大きい番号のレジスタにロードされます。レジスタリストで PC を指定した場合、この命令によって、PC にロードされたアドレス（データ）への分岐が発生します。

エンコード T2 では、1 つのレジスタしか含まないリストはサポートされません。LDMIA 命令のレジスタリストに <Rt> しか含まれていない場合、この命令が Thumb にアセンブルされ、エンコード T1 が使用できない場合は、LDR<c><q> <Rt>,[<Rn>]{,#-4} と等価な命令にアセンブルされます。

リストに SP を含めることはできません。

リストに PC が含まれている場合は、LR をリストに含めることはできず、この命令は IT ブロックの外部に配置するか、IT ブロック内部の最後の命令にする必要があります。

LDMIA および LDMFD は、LDM と同義語です。LDMFD は、フル下降スタックからデータをポップするために使用することを意味しています。

UAL 以前の構文の LDM<c>IA および LDM<c>FD は、LDM<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];

    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);

    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
```

例外

UsageFault、MemManage、BusFault

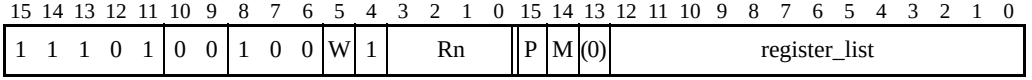
A6.7.41 LDMDB / LDMEA

LDMDB、LDMEA（プリデクリメント複数ロード（空上昇複数ロード））は、ベースレジスタからのアドレスを使用して、連続するメモリ位置から複数のレジスタをロードします。連続するメモリ位置は、このアドレスの 1 つ前で終了します。オプションとして、ロードする最初のメモリ位置のアドレスをベースアドレスに書き戻すことができます。

ロードするレジスタには、PC を含めることができます。この場合、PC にロードされるワードは、アドレスまたは例外の戻り値として扱われ、分岐が発生します。ビット <0> は、Thumb 状態実行への分岐に関する ARM アーキテクチャのインターワーキング規則に準拠するために、1 に設定する必要があります。ビット <0> が 0 の場合、UsageFault 例外が発生します。

エンコード T1 ARMv6T2、ARMv7

LDMDB<c> <Rn>{!},<registers>




```
n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

アセンブラ構文

```
LDMDB<c><q>  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　ベースレジスタを指定します。SP を指定できます。

!　命令によって、変更された値が <Rn> に書き戻されます。! を省略した場合は、命令によって <Rn> は変更されません（ただし、<Rn> が <registers> に含まれる場合は、値がロードされたときにこのレジスタが変更されます）。

<registers>

1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストで、ロードするレジスタのセットを指定します。最下位のメモリアドレスに置かれているデータは最も小さい番号のレジスタにロードされ、以下順に、最上位のメモリアドレスのデータは最も大きい番号のレジスタにロードされます。レジスタリストで PC を指定した場合、この命令によって、PC にロードされたアドレス（データ）への分岐が発生します。

エンコード T1 では、1 つのレジスタしか含まないリストはサポートされません。LDMDB 命令で、このリストに <Rt> のみを指定し、Thumb にアセンブルされた場合は、LDR<c><q> <Rt>,[<Rn>,#-4]{!} と等価な命令にアセンブルされます。

リストに SP を含めることはできません。

リストに PC が含まれている場合は、LR をリストに含めることはできず、この命令は IT ブロックの外部に配置するか、IT ブロック内部の最後の命令にする必要があります。

LDMEA は LDMDB の同義語で、空上昇スタックからデータをポップするために使用されることを示します。

UAL 以前の構文の LDM<c>DB および LDM<c>EA は、LDMDB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);

    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);

    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
```

例外

UsageFault、MemManage、BusFault

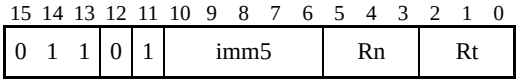
A6.7.42 LDR（イミディエート）

LDR（レジスタロード（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ワードをロードし、レジスタに書き込みます。オフセットアドレシング、ポストインデクスアドレシング、またはプリインデクスアドレシングを使用することができます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

ロードされるレジスタには PC を指定できます。この場合、PC にロードされるワードはアドレスまたは例外の戻り値として扱われ、分岐が発生します。ビット <0> は、Thumb 状態実行への分岐に関する ARM アーキテクチャのインターワーキングルールに準拠するために、1 に設定する必要があります。ビット <0> が 0 の場合は、UsageFault 例外が発生します。

エンコード T1　　ARMv4T、ARMv5T*、ARMv6*、ARMv7

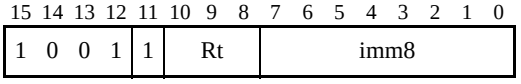
LDR<c> <Rt>, [<Rn>{,#<imm5>}]




```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T2　　ARMv4T、ARMv5T*、ARMv6*、ARMv7

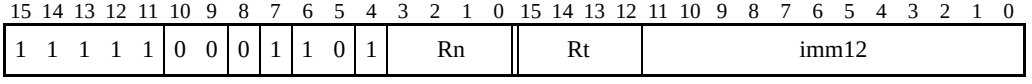
LDR<c> <Rt>,[SP{,#<imm8>}]




```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T3　　ARMv6T2、ARMv7

LDR<c>.W <Rt>,[<Rn>{,#<imm12>}]



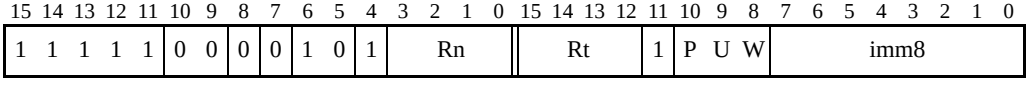

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード T4　　ARMv6T2、ARMv7

LDR<c> <Rt>,[<Rn>,#-<imm8>]

LDR<c> <Rt>,[<Rn>],#+/-<imm8>

LDR<c> <Rt>,[<Rn>,#+/-<imm8>]!




```
if Rn == '1111' then SEE LDR (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRT;
if Rn == '1101' && P == '0' && U == '1' && W == '1' && imm8 == '00000100' then SEE POP;
```

```
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```

アセンブラ構文

LDR<c><q> <Rt>, [<Rn> {, #+/-<imm>}]	オフセット：index==TRUE、wback==FALSE
LDR<c><q> <Rt>, [<Rn>, #+/-<imm>]!	プリインデクス：index==TRUE、wback==TRUE
LDR<c><q> <Rt>, [<Rn>], #+/-<imm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> デスティネーションレジスタを指定します。SP を指定できます。命令が IT ブロック外部に配置されている場合、または IT ブロックの最後の命令の場合は、PC を指定できます。PC を指定した場合は、PC にロードされたアドレス（データ）への分岐が発生します。

<Rn> ベースレジスタを指定します。SP を指定できます。このレジスタが PC の場合については、P.A6-91「*LDR（リテラル）*」を参照して下さい。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレスを生成するために <Rn> の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値値の範囲は、エンコード T1 では 4 の倍数で 0 ～ 124、エンコード T2 では 4 の倍数で 0 ～ 1020、エンコード T3 では 0 ～ 4095 の任意の値、エンコード T4 では 0 ～ 255 の任意の値です。オフセットアドレシング構文では <imm> を省略可能で、この場合はオフセットが 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    else
        R[t] = data;
```

例外

UsageFault、MemManage、BusFault

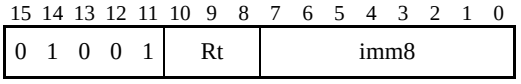
A6.7.43 LDR（リテラル）

LDR（レジスタロード（リテラル））は、PC の値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ワードをロードし、レジスタに書き込みます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

ロードするレジスタとして PC を指定できます。この場合、PC にロードされるワードは、アドレスまたは例外の戻り値として扱われ、分岐が発生します。ビット <0> は、Thumb 状態実行への分岐に関する ARM アーキテクチャのインターワーキングルールに準拠するために、1 に設定する必要があります。ビット <0> が 0 の場合は、UsageFault 例外が発生します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

LDR<c> <Rt>,<label>

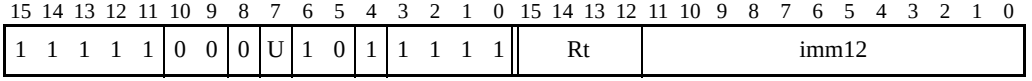



```
t = UInt(Rt);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

エンコード T2 ARMv6T2、ARMv7

LDR<c>.W <Rt>,<label>

LDR<c>.W <Rt>,[PC,#-0] 特殊なケース




```
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

アセンブラ構文

LDR<c><q> <Rt>, <label> 通常の構文

LDR<c><q> <Rt>, [PC, #+/-<imm>] 代替構文

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> デスティネーションレジスタを指定します。SP を指定できます。命令が IT ブロック外部に配置されている場合、または IT ブロックの最後の命令の場合は、PC も指定できます。PC を指定した場合は、PC にロードされたアドレス（データ）への分岐が発生します。

<label> <Rt> にロードされるリテラルデータ項目のラベルを指定します。アセンブラは、この命令の Align(PC,4) の値からこのラベルまでのオフセットに必要となる値を計算します。

オフセットが正の場合は、imm32 はオフセットと等しく、add== TRUE のエンコード T1 および T2 が許可されます。オフセットに許容される値は、エンコード T1 では 4 の倍数で 0 ～ 1020、エンコード T2 では 0 ～ 4095 の任意の値です。

オフセットが負の場合はエンコード T2 が許可され、imm32 は負のオフセットと等しく、add== FALSE になります。オフセットに許容される値は -4095 ～ -1 です。

代替構文形式の場合、次のオプションが指定できます。

+/- イミディエートオフセットを Align(PC, 4) の値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために命令の Align(PC, 4) の値に対して加算または減算されるイミディエートオフセットを指定します。オフセットに許容される値は、エンコード T1 では 4 の倍数で 0 ～ 1020、エンコード T2 では 0 ～ 4095 までの任意の値です。

注

可能であれば、代替構文形式は使用しないことをお勧めします。ただし、エンコード T2 で U ビットおよびすべてのイミディエートビットが 0 の場合、可能な構文は LDR<c><q> <Rt>, [PC,#-0] のみです。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,4];
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    else
        R[t] = data;
```

例外

UsageFault、MemManage、BusFault

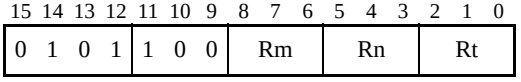
A6.7.44 LDR（レジスタ）

LDR（レジスタロード（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 ワードをロードし、レジスタに書き込みます。オフセットレジスタの値は、左に 0、1、2、または 3 ビットシフトすることができます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

ロードされるレジスタには PC を指定できます。この場合、PC にロードされるワードは、アドレスまたは例外の戻り値として扱われ、分岐が発生します。ビット <0> は、Thumb 状態実行への分岐に関する ARM アーキテクチャのインターワーキングルールに準拠するために、1 に設定する必要があります。ビット <0> が 0 の場合は、UsageFault 例外が発生します。

エンコード T1　　ARMv4T、ARMv5T*、ARMv6*、ARMv7

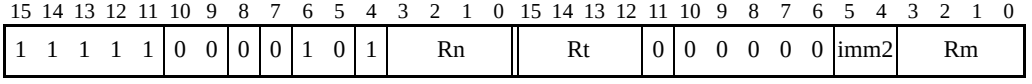
LDR<c> <Rt>,[<Rn>,<Rm>]




```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2　　ARMv6T2、ARMv7

LDR<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]




```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(m) then UNPREDICTABLE;
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

アセンブラ構文

```
LDR<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt>　デスティネーションレジスタを指定します。SP を指定できます。命令が IT ブロック外部に配置されている場合、または IT ブロックの最後の命令の場合は、PC も指定できます。PC を指定した場合は、PC にロードされたアドレス（データ）への分岐が発生します。

<Rn>　ベース値を含むレジスタを指定します。SP を指定できます。

<Rm>　アドレス生成のために左にシフトして <Rn> の値に加算されるオフセットが含まれます。

<shift>　<Rm> からの値を左にシフトするビット数を指定します。範囲は 0 ～ 3 です。このオプションを省略した場合は、0 ビットのシフトと見なされ、両方のエンコードが許可されます。このオプションを指定した場合は、エンコード T2 のみが許可されます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    else
        R[t] = data;
```

例外

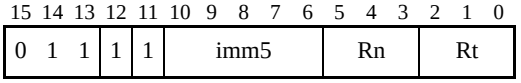
UsageFault、MemManage、BusFault

A6.7.45 LDRB（イミディエート）

LDRB（レジスタロード バイト（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。オフセットアドレシング、ポストインデクスアドレシング、またはプリインデクスアドレシングを使用できます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

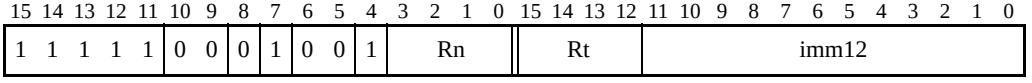
LDRB<c> <Rt>,[<Rn>{,#<imm5>}]




```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T2 ARMv6T2、ARMv7

LDRB<c>.W <Rt>,[<Rn>{,#<imm12>}]



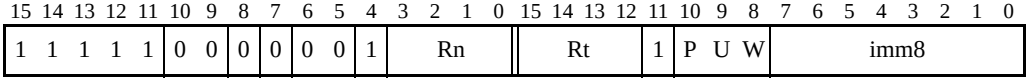

```
if Rt == '1111' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

LDRB<c> <Rt>,[<Rn>,#-<imm8>]

LDRB<c> <Rt>,[<Rn>],#+/-<imm8>

LDRB<c> <Rt>,[<Rn>,#+/-<imm8>]!




```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRBT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```


アセンブラ構文

LDRB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]	オフセット：index==TRUE、wback==FALSE
LDRB<c><q> <Rt>, [<Rn>, #+/-<imm>]!	プリインデクス：index==TRUE、wback==TRUE
LDRB<c><q> <Rt>, [<Rn>], #+/-<imm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> デスティネーションレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。このレジスタが PC の場合については、P.A6-97「*LDRB（リテラル）*」を参照して下さい。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために <Rn> の値に加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は、エンコード T1 の場合は 0 ～ 31、エンコード T2 の場合は 0 ～ 4095、エンコード T3 の場合は 0 ～ 255 です。オフセットアドレシング構文では <imm> は省略可能で、その場合はオフセットが 0 と見なされます。

UAL 以前の構文の LDR<c>B は、LDRB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

例外

MemManage、BusFault

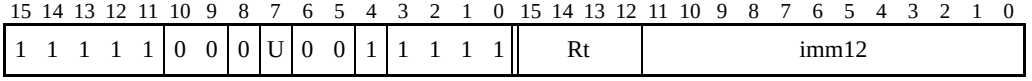
A6.7.46 LDRB（リテラル）

LDRB（レジスタロード バイト（リテラル））は、PC の値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

LDRB<c> <Rt>,<label>

LDRB<c> <Rt>,[PC,#-0] 特殊なケース



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	U	0	0	1	1	1	1	1	Rt				imm12											

```
if Rt == '1111' then SEE PLD;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```


アセンブラ構文

LDRB<c><q> <Rt>, <label>	通常の形式
LDRB<c><q> <Rt>, [PC, #+/-<imm>]	代替形式

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> デスティネーションレジスタを指定します。

<label> <Rt> にロードされるリテラルデータ項目のラベルを指定します。アセンブラは、この命令の Align(PC,4) の値からこのラベルまでのオフセットに必要な値を計算します。

オフセットが正の場合はエンコード T1 が許可され、imm32 はオフセットと等しく、add== TRUE になります。オフセットに許容される値の範囲は 0 ～ 4095 です。

オフセットが負の場合はエンコード T1 が許可され、imm32 は負のオフセットと等しく、add== FALSE になります。オフセットに許容される値の範囲は -4095～-1 です。

代替構文形式の場合、次のパラメータが使用されます。

+/- イミディエートオフセットを Align(PC, 4) の値に加算する場合 (add== TRUE) は、+ を指定するか、省略します。減算する場合 (add== FALSE) は - を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために命令の Align(PC, 4) の値に対して加算または減算されるイミディエートオフセットを指定します。

許容される値の範囲は 0 ～ 4095 です。

注

可能であれば、代替構文形式は使用しないことをお勧めします。ただし、エンコード T1 で、U ビットとすべてのイミディエートビットが 0 の場合、使用できる構文は LDRB<c><q> <Rt>, [PC,#-0] のみです。

UAL 以前の構文の LDR<c>B は、LDRB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = ZeroExtend(MemU[address,1], 32);
```

例外

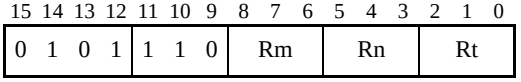
MemManage、BusFault

A6.7.47 LDRB（レジスタ）

LDRB（レジスタロード バイト（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。オフセットレジスタの値は、左に 0、1、2、または 3 ビットシフトすることができます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

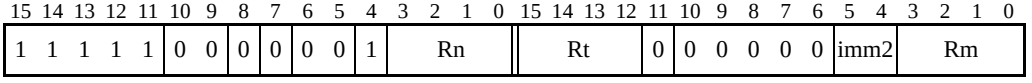
LDRB<c> <Rt>,[<Rn>,<Rm>]




```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

LDRB<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]




```
if Rt == '1111' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || BadReg(m) then UNPREDICTABLE;
```


アセンブラ構文

```
LDRB<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　ベース値を含むレジスタを指定します。SP を指定できます。

<Rm>　アドレス生成のために左にシフトして <Rn> の値に加算されるオフセットが含まれます。

<shift>　<Rm> からの値を左にシフトするビット数を指定します。範囲は 0 ～ 3 です。このオプションを省略した場合、0 ビットのシフトと見なされ、両方のエンコードが許可されます。このオプションを指定した場合は、エンコード T2 のみが許可されます。

UAL 以前の構文の LDR<c>B は、LDRB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1],32);
```

例外

MemManage、BusFault

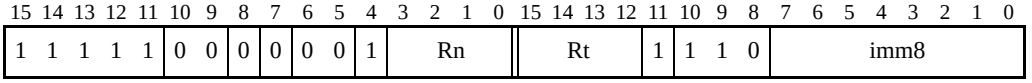
A6.7.48 LDRBT

LDRBT（非特権レジスタロード バイト）は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

プロセッサが非特権状態で実行している場合と同様にメモリアクセスが制限されます(実際にプロセッサが権限のない状態で実行している場合と違いはありません)。

エンコード T1 ARMv6T2、ARMv7

LDRBT<c> <Rt>,[<Rn>,#<imm8>]




```
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```


アセンブラ構文

```
LDRBT<c><q>  <Rt>, [<Rn> {, #<imm>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt>　デスティネーションレジスタを指定します。

<Rn>　ベースレジスタを指定します。SP を指定できます。

<imm>　アドレス生成のために <Rn> の値に加算されるイミディエートオフセットを指定します。許容される値の範囲は 0 ～ 255 です。<imm> を省略した場合、オフセットは 0 と見なされます。

UAL 以前の構文の LDR<c>BT は、LDRBT<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    R[t] = ZeroExtend(MemU_unpriv[address,1],32);
```

例外

MemManage、BusFault

A6.7.49 LDRD（イミディエート）

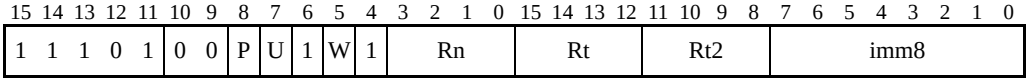
この命令（レジスタロード デュアル（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 2 ワードをロードし、2 つのレジスタに書き込みます。オフセットアドレシング、ポストインデクスアドレシング、プリインデクスアドレシングを使用することができます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

LDRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm8>}]

LDRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm8>

LDRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm8>]!




```
if P == '0' && W == '0' then SEE "Related encodings";
if Rn == '1111' then SEE LDRD (literal);
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if BadReg(t) || BadReg(t2) || t == t2 then UNPREDICTABLE;
```

関連エンコード P.A5-21「*ダブルワードのロード/ ストア、排他ロード/ ストア、テーブル分岐*」を参照

アセンブラ構文

LDRD<c><q> <Rt>,<Rt2>,[<Rn>{,#+/-<imm>}]	オフセット：index==TRUE、wback==FALSE
LDRD<c><q> <Rt>,<Rt2>,[<Rn>,#+/-<imm>]!	プリインデクス：index==TRUE、wback==TRUE
LDRD<c><q> <Rt>,<Rt2>,[<Rn>],#+/-<imm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> 最初のデスティネーションレジスタを指定します。

<Rt2> 2 番目のデスティネーションレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。オフセットアドレシングの構文形式では、PC も指定できます。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために <Rn> の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は、4 の倍数で 0 ～ 1020 です。オフセットアドレシング構文では <imm> を省略可能で、この場合オフセットは 0 と見なされます。

UAL 以前の構文の LDR<c>D は、LDRD<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = MemA[address,4];
    R[t2] = MemA[address+4,4];
    if wback then R[n] = offset_addr;
```

例外

UsafeFault、MemManage、BusFault

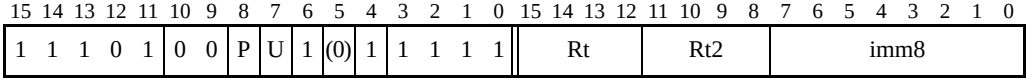
A6.7.50 LDRD（リテラル）

LDRD（レジスタロードデュアル（リテラル））は、PC の値とイミディエートオフセットからアドレスを計算して、そのメモリから 2 ワードをロードし、2 つのレジスタに書き込みます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

LDRD<c> <Rt>,<Rt2>,<label>

LDRD<c> <Rt>,<Rt2>,[PC,#-0] 特殊なケース



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	P	U	1	(0)	1	1	1	1	1	Rt				Rt2				imm8							

```
if P == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);
imm32 = ZeroExtend(imm8:'00', 32);  add = (U == '1');
if BadReg(t) || BadReg(t2) || t == t2 then UNPREDICTABLE;
```

関連エンコード P.A5-21「*ダブルワードのロード/ストア、排他ロード/ストア、テーブル分岐*」を参照

アセンブラ構文

LDRD<c><q> <Rt>, <Rt2>, <label>	通常の形式
LDRD<c><q> <Rt>, <Rt2>, [PC, #+/-<imm>]	代替形式

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> 最初のデスティネーションレジスタを指定します。

<Rt2> 2 番目のデスティネーションレジスタを指定します。

<label> <Rt> にロードされるリテラルデータ項目のラベル。アセンブラは、この命令の Align(PC,4) の値からラベルまでのオフセットに必要な値を計算します。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために <Rn> の値に加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は、4 の倍数で 0 ～ 1020 です。オフセットアドレシング構文では、<imm> を省略可能で、この場合オフセットは 0 と見なされます。

注

可能であれば、代替構文形式は使用しないことをお勧めします。ただし、U ビットとすべてのイミディエートビットが 0 の場合、使用可能な構文は LDRD<c><q> <Rt>, [PC,#-0] のみです。

UAL 以前の構文の LDR<c>D は、LDRD<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    R[t] = MemA[address,4];
    R[t2] = MemA[address+4,4];
```

例外

UsageFault、MemManage、BusFault

A6.7.51 LDREX

LDREX（排他レジスタロード）は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ワードをロードしてレジスタに書き込み、次の動作を実行します。

- アドレスに共有可メモリ属性が設定されている場合は、実行中のプロセッサのための排他アクセスとしてその物理アドレスをグローバルモニタ中にマークします。
- 実行中のプロセッサがローカルモニタ中にアクティブな排他アクセスを示すようにさせます。

メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

LDREX<c> <Rt>,[<Rn>{,#<imm8>}]


```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if BadReg(t) || n == 15 then UNPREDICTABLE;
```

アセンブラ構文

```
LDREX<c><q>  <Rt>, [<Rn> {,#<imm>}]
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> デスティネーションレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。

<imm> アドレス生成のために <Rn> の値に加算されるイミディエートオフセットを指定します。許容される値の範囲は、4 の倍数で 0 ～ 1020 です。<imm> は省略可能で、その場合オフセットは 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    SetExclusiveMonitors(address,4);
    R[t] = MemA[address,4];
```

例外

UsageFault、MemManage、BusFault

A6.7.52 LDREXB

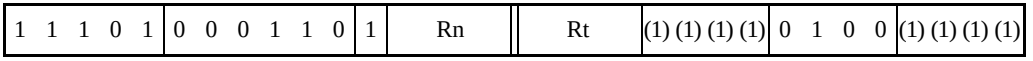
LDREXB（排他レジスタロード バイト）は、ベースレジスタの値からアドレスを抽出して、そのメモリから 1 バイトをロードし、32 ビットワードにゼロ拡張してレジスタに書き込み、次の動作を実行します。

- アドレスに共有可メモリ属性が設定されている場合は、実行中のプロセッサのための排他アクセスとしてその物理アドレスをグローバルモニタ中にマークします。
- 実行中のプロセッサがローカルモニタ中にアクティブな排他アクセスを示すようにさせます。

メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6K またはそれ以降のすべてのバージョンの Thumb ISA

LDREXB<c> <Rt>, [<Rn>]




```
t = UInt(Rt);  n = UInt(Rn);
if BadReg(t) || n == 15 then UNPREDICTABLE;
```

アセンブラ構文

```
LDREXB<c><q>  <Rt>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> デスティネーションレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    SetExclusiveMonitors(address,1);
    R[t] = ZeroExtend(MemA[address,1], 32);
```

例外

MemManage、BusFault

A6.7.53 LDREXH

LDREXH（排他レジスタロード ハーフワード）は、ベースレジスタの値からアドレスを抽出して、そのメモリから 1 ハーフワードをロードし、32 ビットワードにゼロ拡張してレジスタに書き込み、次の動作を実行します。

- アドレスに共有可メモリ属性が設定されている場合は、実行中のプロセッサのための排他アクセスとしてその物理アドレスをグローバルモニタ中にマークします。
- 実行中のプロセッサがローカルモニタ中にアクティブな排他アクセスを示すようにさせます。

メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6K またはそれ以降のすべてのバージョンの Thumb ISA

LDREXH<c> <Rt>, [<Rn>]


```
t = UInt(Rt);  n = UInt(Rn);
if BadReg(t) || n == 15 then UNPREDICTABLE;
```


アセンブラ構文

```
LDREXH<c><q>  <Rt>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt>　　デスティネーションレジスタを指定します。

<Rn>　　ベースレジスタを指定します。SP を指定できます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    SetExclusiveMonitors(address,2);
    R[t] = ZeroExtend(MemA[address,2], 32);
```

例外

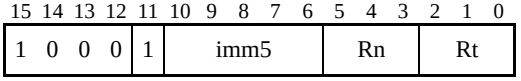
UsageFault、MemManage、BusFault

A6.7.54 LDRH（イミディエート）

LDRH（レジスタロード ハーフワード（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。オフセットアドレシング、ポストインデクスアドレシング、プリインデクスアドレシングを使用できます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

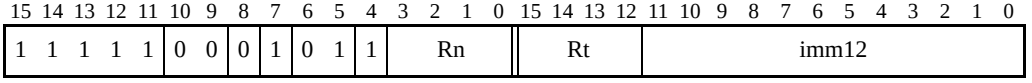
LDRH<c> <Rt>,[<Rn>{,#<imm5>}]




```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T2 ARMv6T2、ARMv7

LDRH<c>.W <Rt>,[<Rn>{,#<imm12>}]



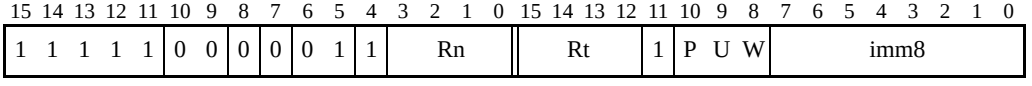

```
if Rt == '1111' then SEE "Unallocated memory hints";
if Rn == '1111' then SEE LDRH (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

LDRH<c> <Rt>,[<Rn>,#-<imm8>]

LDRH<c> <Rt>,[<Rn>],#+/-<imm8>

LDRH<c> <Rt>,[<Rn>,#+/-<imm8>]!




```
if Rn == '1111' then SEE LDRH (literal);
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Unallocated memory hints";
if P == '1' && U == '1' && W == '0' then SEE LDRHT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```


アセンブラ構文

LDRH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]	オフセット：index==TRUE、wback==FALSE
LDRH<c><q> <Rt>, [<Rn>, #+/-<imm>]!	プリインデクス：index==TRUE、wback==TRUE
LDRH<c><q> <Rt>, [<Rn>], #+/-<imm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> デスティネーションレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。このレジスタが PC の場合については、P.A6-113「*LDRH（リテラル）*」を参照して下さい。

+/- イミディエートオフセットをベースレジスタ値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために <Rn> の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は、エンコード T1 では 2 の倍数で 0 ～ 62、エンコード T2 では 0 ～ 4095 までの任意の値、エンコード T3 では 0 ～ 255 までの任意の値です。オフセットアドレシング構文では <imm> を省略可能で、この場合オフセットは 0 と見なされます。

UAL 以前の構文の LDR<c>H は、LDRH<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

例外

UsageFault、MemManage、BusFault

未割り当てメモリヒント

エンコード T2 の Rt フィールドが "1111" の場合、または、エンコード T3 の Rt フィールド、P ビット、U ビット、W ビットがそれぞれ "1111"、"1"、"0"、"0" の場合、この命令は未割り当てメモリヒントです。

未割り当てメモリヒントは、NOP として実装する必要があります。ソフトウェアでは使用しないで下さい。そのため、これらの命令の UAL アセンブラ構文はありません。

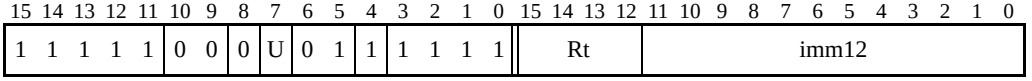
A6.7.55 LDRH（リテラル）

LDRH（レジスタロード ハーフワード（リテラル））は、PC の値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

LDRH<c> <Rt>,<label>

LDRH<c> <Rt>,[PC,#-0] 特殊なケース




```
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```


アセンブラ構文

LDRH<c><q> <Rt>, <label>	通常の形式
LDRH<c><q> <Rt>, [PC, #+/-<imm>]	代替形式

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<label> <Rt> にロードされるリテラルデータ項目のラベルを指定します。アセンブラは、ADR 命令の Align(PC,4) の値からこのラベルまでのオフセットに必要な値を計算します。

オフセットが正の場合、エンコード T1 が許可され、imm32 はオフセットと等しく、add == TRUE です。オフセットに許容される値の範囲は 0 ～ 4095 です。

オフセットが負の場合、エンコード T1 が許可され、imm32 は負のオフセットと等しく、add == FALSE です。オフセットに許容される値の範囲は -4095 ～ -1 です。

代替構文形式の場合、次のオプションが使用されます。

+/- イミディエートオフセットを Align(PC, 4) の値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために命令の Align(PC, 4) の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は 0 ～ 4095 です。

注

可能であれば、代替構文形式は使用しないことをお勧めします。ただし、T1 エンコードで U ビットとすべてのイミディエートビットが 0 の場合、使用できる構文は LDRH<c><q> <Rt>,[PC,#0] のみです。

UAL 以前の構文の LDR<c>H は、LDRH<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = ZeroExtend(data, 32);
```

例外

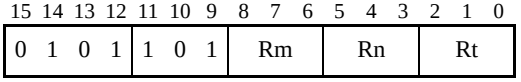
UsageFault、MemManage、BusFault

A6.7.56 LDRH（レジスタ）

LDRH（レジスタロード ハーフワード（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。オフセットレジスタの値は、左に 0、1、2、または 3 ビットシフトすることができます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

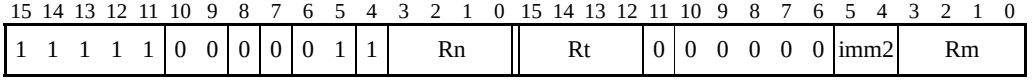
LDRH<c> <Rt>,[<Rn>,<Rm>]




```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

LDRH<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]




```
if Rn == '1111' then SEE LDRH (literal);
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || BadReg(m) then UNPREDICTABLE;
```


アセンブラ構文

```
LDRH<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　ベース値を含むレジスタを指定します。SP を指定できます。

<Rm>　アドレス生成のために、左にシフトして <Rn> の値に加算されるオフセットが含まれます。

<shift>　<Rm> からの値を左にシフトするビット数を指定します。範囲は 0 ～ 3 です。このオプションを省略した場合は、0 ビットのシフトと見なされ、両方のエンコードが許可されます。このオプションを指定した場合は、エンコード T2 のみが許可されます。

UAL 以前の構文の LDR<c>H は、LDRH<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

例外

UsageFault、MemManage、BusFault

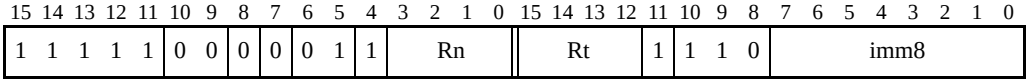
A6.7.57 LDRHT

LDRHT（非特権レジスタロード ハーフワード）は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードにゼロ拡張してレジスタに書き込みます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

プロセッサが非特権状態で実行している場合と同様にメモリアクセスが制限されます(実際にプロセッサが非特権状態で実行している場合と違いはありません)。

エンコード T1 ARMv6T2、ARMv7

LDRHT<c> <Rt>,[<Rn>,#<imm8>]




```
if Rn == '1111' then SEE LDRH (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```


アセンブラ構文

```
LDRHT<c><q>  <Rt>, [<Rn> {, #<imm>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt>　デスティネーションレジスタを指定します。

<Rn>　ベースレジスタを指定します。SP を指定できます。

<imm>　アドレス生成のために <Rn> の値に加算されるイミディエートオフセットを指定します。許容される値の範囲は 0 ～ 255 です。<imm> は省略可能で、その場合オフセットは 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    data = MemU_unpriv[address,2];
    R[t] = ZeroExtend(data, 32);
```

例外

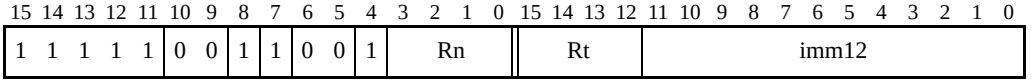
UsageFault、MemManage、BusFault

A6.7.58 LDRSB（イミディエート）

LDRSB（レジスタロード符号付きバイト（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。オフセットアドレシング、ポストインデクスアドレシング、プリインデクスアドレシングを使用できます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

LDRSB<c> <Rt>,[<Rn>,#<imm12>]




```
if Rt == '1111' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

LDRSB<c> <Rt>,[<Rn>,#-<imm8>]

LDRSB<c> <Rt>,[<Rn>],#+/-<imm8>

LDRSB<c> <Rt>,[<Rn>,#+/-<imm8>]!


```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRSBT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```


アセンブラ構文

LDRSB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]	オフセット：index==TRUE、wback==FALSE
LDRSB<c><q> <Rt>, [<Rn>, #+/-<imm>]!	プリインデクス：index==TRUE、wback==TRUE
LDRSB<c><q> <Rt>, [<Rn>], #+/-<imm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> デスティネーションレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。PC を指定した場合については、P.A6-121「*LDRSB（リテラル）*」を参照して下さい。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- に指定します #0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために <Rn> の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は、エンコード T1 では 0 ～ 4095、エンコード T2 では 0 ～ 255 です。オフセットアドレシング構文では <imm> を省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の LDR<c>SB は、LDRSB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

例外

MemManage、BusFault

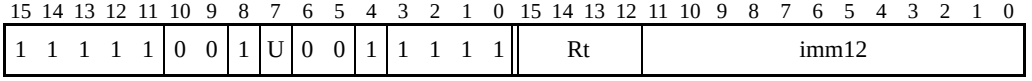
A6.7.59 LDRSB（リテラル）

LDRSB（レジスタロード符号付きバイト（リテラル））は、PC の値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

LDRSB<c> <Rt>,<label>

LDRSB<c> <Rt>,[PC,#-0] 特殊なケース




```
if Rt == '1111' then SEE PLI;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```


アセンブラ構文

LDRSB<c><q> <Rt>, <label>	通常の形式
LDRSB<c><q> <Rt>, [PC, #+/-<imm>]	代替形式

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<label> <Rt> にロードされるリテラルデータ項目のラベルを指定します。アセンブラは、ADR 命令の Align(PC,4) の値からこのラベルまでのオフセットに必要な値を計算します。

オフセットが正の場合は、エンコード T1 が許可され、imm32 はオフセットと等しく、add == TRUE です。オフセットに許容される値の範囲は 0 ～ 4095 です。

オフセットが負の場合は、エンコード T1 が許可され、imm32 は負のオフセットと等しく、add == FALSE です。オフセットに許容される値の範囲は -4095 ～ -1 です。

代替構文形式の場合、次のオプションが使用されます。

+/- イミディエートオフセットを Align(PC, 4) の値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために命令の Align(PC,4) の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は 0 ～ 4095 です。

注

可能であれば、代替構文形式は使用しないことをお勧めします。ただし、エンコード T1 で U ビットとすべてのイミディエートビットが 0 の場合、使用可能な構文は LDRSB<c><q> <Rt>,[PC,#0] のみです。

UAL 以前の構文の LDR<c>SB は、LDRSB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = SignExtend(MemU[address,1], 32);
```

例外

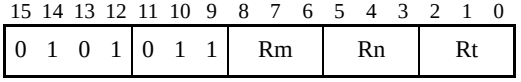
MemManage、BusFault

A6.7.60 LDRSB（レジスタ）

LDRSB（レジスタロード符号付きバイト（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。オフセットレジスタの値は、左に 0、1、2、または 3 ビットシフトすることができます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

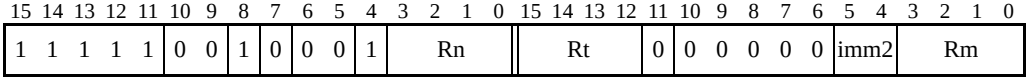
LDRSB<c> <Rt>,[<Rn>,<Rm>]




```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

LDRSB<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]




```
if Rt == '1111' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || BadReg(m) then UNPREDICTABLE;
```


アセンブラ構文

```
LDRSB<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　ベース値を含むレジスタを指定します。SP を指定できます。

<Rm>　<Rn>アドレス生成のために左にシフトして<Rn>の値に加算されるオフセットが含まれます。

<shift>　<Rm> からの値を左にシフトするビット数を指定します。範囲は 0 ～ 3 です。このオプションを省略した場合は、0 ビットのシフトと見なされ、両方のエンコードが許可されます。このオプションを指定した場合は、T2 エンコードのみが許可されます。

UAL 以前の構文の LDR<c>SB は、LDRSB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
```

例外

MemManage、BusFault

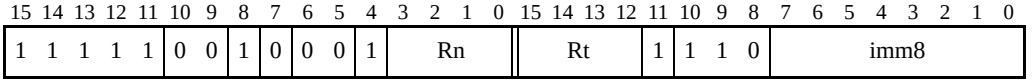
A6.7.61 LDRSBT

LDRSBT（非特権レジスタロード符号付きバイト）は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 バイトをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

プロセッサが非特権状態で実行している場合と同様にメモリアクセスが制限されます(実際にプロセッサが非特権状態で実行している場合と違いはありません)。

エンコード T1 ARMv6T2、ARMv7

LDRSBT<c> <Rt>,[<Rn>,#<imm8>]




```
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```


アセンブラ構文

```
LDRSBT<c><q>  <Rt>, [<Rn> {, #<imm>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt>　デスティネーションレジスタを指定します。

<Rn>　ベースレジスタを指定します。SP を指定できます。

<imm>　アドレス生成のために <Rn> の値に加算されるイミディエートオフセットを指定します。許容される値の範囲は 0 ～ 255 です。<imm> は省略可能で、その場合オフセットは 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    R[t] = SignExtend(MemU_unpriv[address,1], 32);
```

例外

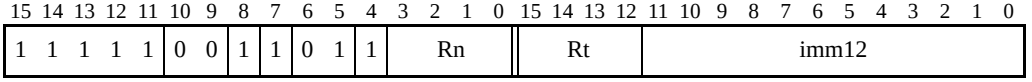
MemManage、BusFault

A6.7.62 LDRSH（イミディエート）

LDRSH（レジスタロード符号付きハーフワード（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。オフセットアドレシング、ポストインデクスアドレシング、プリインデクスアドレシングを使用できます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

LDRSH<c> <Rt>,[<Rn>,#<imm12>]



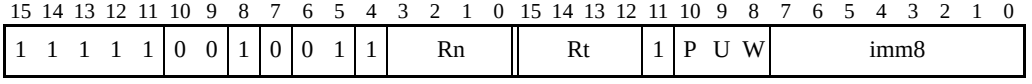

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

LDRSH<c> <Rt>,[<Rn>,#-<imm8>]

LDRSH<c> <Rt>,[<Rn>],#+/-<imm8>

LDRSH<c> <Rt>,[<Rn>,#+/-<imm8>]!




```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Unallocated memory hints";
if P == '1' && U == '1' && W == '0' then SEE LDRSHT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```


アセンブラ構文

LDRSH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]	オフセット：index==TRUE、wback==FALSE
LDRSH<c><q> <Rt>, [<Rn>, #+/-<imm>]!	プリインデクス：index==TRUE、wback==TRUE
LDRSH<c><q> <Rt>, [<Rn>], #+/-<imm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> デスティネーションレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。PC を指定した場合については、P.A6-129「*LDRSH（リテラル）*」を参照して下さい。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0.

<imm> アドレス生成のために <Rn> の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は、エンコード T1 では 0 ～ 4095、エンコード T2 では 0 ～ 255 です。オフセットアドレシング構文では <imm> を省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の LDR*LDR<c>SH* は、LDRSH<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

例外

UsageFault、MemManage、BusFault

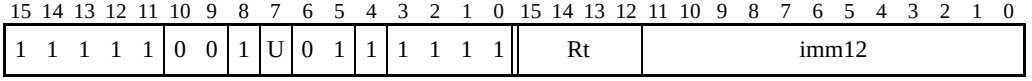
A6.7.63 LDRSH（リテラル）

LDRSH（レジスタロード符号付きハーフワード（リテラル））は、PC の値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

LDRSH<c> <Rt>,<label>

LDRSH<c> <Rt>,[PC,#-0] 特殊なケース




```
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

アセンブラ構文

LDRSH<c><q> <Rt>, <label> 通常の形式

LDRSH<c><q> <Rt>, [PC, #+/-<imm>] 代替形式

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<label> <Rt> にロードされるリテラルデータ項目のラベルを指定します。アセンブラは、ADR 命令の Align(PC,4) の値からこのラベルまでのオフセットに必要な値を計算します。

オフセットが正の場合はエンコード T1 が許可され、imm32 はオフセットと等しく、add == TRUE です。オフセットに許容される値の範囲は 0 ～ 4095 です。

オフセットが負の場合はエンコード T1 が許可され、imm32 は負のオフセットと等しく、add == FALSE です。オフセットに許容される値の範囲は -4095 ～ -1 です。

代替構文形式の場合、次のオプションが使用されます。

+/- イミディエートオフセットを Align(PC, 4) の値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために命令の Align(PC, 4) の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は 0 ～ 4095 です。

注

可能であれば、代替構文形式は使用しないことをお勧めします。ただし、エンコード T1 で U ビットとすべてのイミディエートビットが 0 の場合、使用可能な構文は LDRSH<c><q> <Rt>,[PC,#0] のみです。

UAL 以前の構文の LDR<c>SH は、LDRSH<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = SignExtend(data, 32);
```

例外

UsageFault、MemManage、BusFault

未割り当てメモリヒント

エンコード T1 の Rt フィールドが "1111" の場合、その命令は未割り当てメモリヒントです。

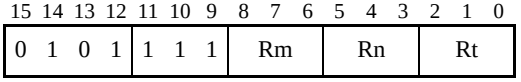
未割り当てメモリヒントは、NOP として実装する必要があります。ソフトウェアでは使用しないで下さい。そのため、この命令には UAL アセンブラ構文はありません。

A6.7.64 LDRSH（レジスタ）

LDRSH（レジスタロード符号付きハーフワード（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。オフセットレジスタの値は、左に 0、1、2、または 3 ビットシフトすることができます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

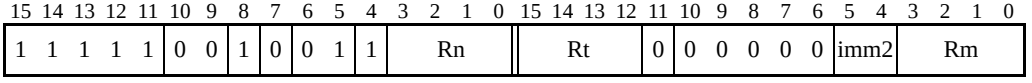
LDRSH<c> <Rt>,[<Rn>,<Rm>]




```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

LDRSH<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]




```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || BadReg(m) then UNPREDICTABLE;
```


アセンブラ構文

```
LDRSH<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt>　デスティネーションレジスタを指定します。

<Rn>　ベース値を含むレジスタを指定します。SP を指定できます。

<Rm>　アドレス生成のために左にシフトして <Rn> の値に加算されるオフセットが含まれます。

<shift>　<Rm> からの値を左にシフトするビット数を指定します。範囲は 0 ～ 3 です。このオプションを省略した場合は、0 ビットのシフトと見なされ、両方のエンコードが許可されます。このオプションを指定した場合は、T2 エンコードのみが許可されます。

UAL 以前の構文の LDR<c>SH は、LDRSH<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

例外

UsageFault、MemManage、BusFault

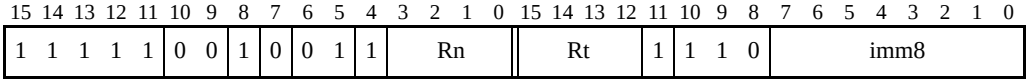
A6.7.65 LDRSHT

LDRSHT（非特権レジスタロード符号付きハーフワード）は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ハーフワードをロードし、32 ビットワードに符号拡張してレジスタに書き込みます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

プロセッサが非特権状態で実行している場合と同様にメモリアクセスが制限されます（実際にプロセッサが非特権状態で実行している場合と違いはありません）。

エンコード T1 ARMv6T2、ARMv7

LDRSHT<c> <Rt>,[<Rn>,#<imm8>]




```
if Rn == '1111' then SEE LDRSH (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

アセンブラ構文

```
LDRSHT<c><q>  <Rt>, [<Rn>, {, #<imm>}]
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> デスティネーションレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。

<imm> アドレス生成のために <Rm> の値に加算されるイミディエートオフセットを指定します。許容される値の範囲は 0 ～ 255 です。<imm> は省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の LDR<c>SH は、LDRSH<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    data = MemU_unpriv[address,2];
    R[t] = SignExtend(data, 32);
```

例外

UsageFault、MemManage、BusFault

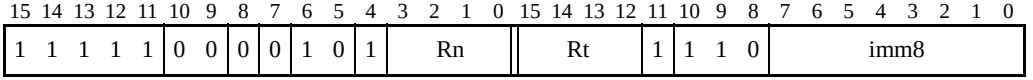
A6.7.66 LDRT

LDRT（非特権レジスタ ロード）は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、そのメモリから 1 ワードをロードし、レジスタに書き込みます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

プロセッサが非特権状態で実行している場合と同様にメモリアクセスが制限されます(実際にプロセッサが非特権状態で実行している場合と違いはありません)。

エンコード T1 ARMv6T2、ARMv7

LDRT<c> <Rt>,[<Rn>,#<imm8>]




```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

アセンブラ構文

```
LDRT<c><q>  <Rt>, [<Rn> {, #<imm>}]
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> デスティネーションレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。

<imm> アドレス生成のために <Rn> の値に加算されるイミディエートオフセットを指定します。許容される値の範囲は 0 ～ 255 です。<imm> は省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の LDR<c>T は、LDRT<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    data = MemU_unpriv[address,4];
    R[t] = data;
```

例外

UsageFault、MemManage、BusFault

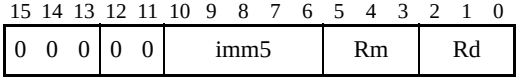
A6.7.67 LSL（イミディエート）

LSL（論理左シフト（イミディエート））は、レジスタの値のビットをイミディエート値で指定されたビット数だけ左にシフト（空いたビットには 0 が入ります）して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

LSLS <Rd>,<Rm>,#<imm5> IT ブロック外部

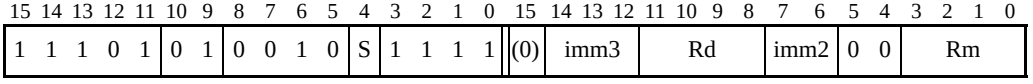
LSL<c> <Rd>,<Rm>,#<imm5> IT ブロック内部




```
if imm5 == '00000' then SEE MOV (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('00', imm5);
```

エンコード T2 ARMv6T2、ARMv7

LSL{S}<c>.W <Rd>,<Rm>,#<imm5>




```
if (imm3:imm2) == '00000' then SEE MOV (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('00', imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
LSL{S}<c><q>  <Rd>, <Rm>, #<imm5>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rm>　　第一オペランドを格納しているレジスタを指定します。

<imm5>　　シフトのビット数を 0 ～ 31 の範囲で指定します。詳細については、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_LSL, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

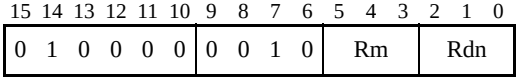
A6.7.68 LSL（レジスタ）

LSL（論理左シフト（レジスタ））は、レジスタの値のビットを可変のビット数だけ左にシフト（空いたビットには 0 が入ります）して、結果をデスティネーションレジスタに書き込みます。シフトする可変のビット数は、レジスタの下位バイトから読み出されます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

LSLS <Rdn>,<Rm> IT ブロック外部

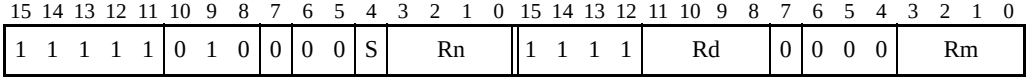
LSL<c> <Rdn>,<Rm> IT ブロック内部




```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

エンコード T2 ARMv6T2、ARMv7

LSL{S}<c>.W <Rd>,<Rn>,<Rm>




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
LSL{S}<c><q>  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　下位バイトにシフトのビット数を含むレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_LSL, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

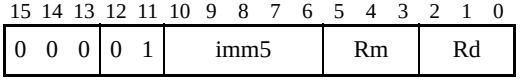
A6.7.69 LSR（イミディエート）

LSR（論理右シフト（イミディエート））は、レジスタの値のビットをイミディエート値で指定されたビット数だけ右にシフト（空いたビットには 0 が入ります）して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

LSRS <Rd>,<Rm>,#<imm5> IT ブロック外部

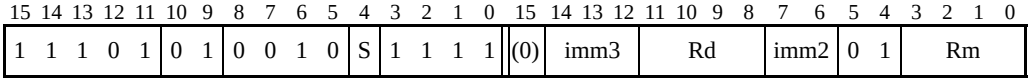
LSR<c> <Rd>,<Rm>,#<imm5> IT ブロック内部




```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('01', imm5);
```

エンコード T2 ARMv6T2、ARMv7

LSR{S}<c>.W <Rd>,<Rm>,#<imm5>




```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('01', imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
LSR{S}<c><q>  <Rd>, <Rm>, #<imm5>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rm>　　第一オペランドを格納しているレジスタを指定します。

<imm5>　　シフトするビット数を 1 ～ 32 の範囲で指定します。詳細については、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_LSR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

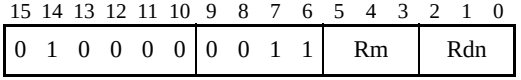
A6.7.70 LSR（レジスタ）

LSR（論理右シフト（レジスタ））は、レジスタの値のビットを可変のビット数だけ右にシフト（空いたビットには 0 が入ります）して、結果をデスティネーションレジスタに書き込みます。シフトする可変のビット数は、レジスタの下位バイトから読み出されます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

LSRS <Rdn>,<Rm> IT ブロック外部

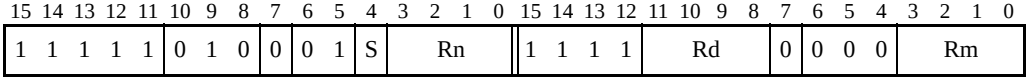
LSR<c> <Rdn>,<Rm> IT ブロック内部




```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

エンコード T2 ARMv6T2、ARMv7

LSR{S}<c>.W <Rd>,<Rn>,<Rm>




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
LSR{S}<c><q>  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　下位バイトにシフトのビット数を含むレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_LSR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

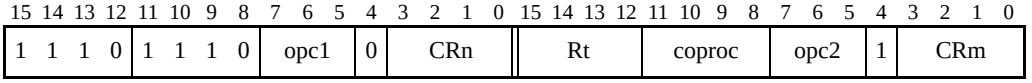
A6.7.71 MCR、MCR2

MCR、MCR2（ARM レジスタからコプロセッサへの移動）は、ARM レジスタの値をコプロセッサに転送します。

コプロセッサが命令を実行できない場合は、UsageFault 例外が発生します。

エンコード T1 すべての Thumb-2 ISA

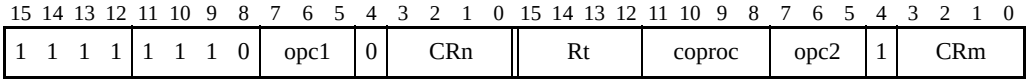
MCR<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}




```
t = UInt(Rt);  cp = UInt(coproc);
if t == 15 || t == 13 then UNPREDICTABLE;
```

エンコード T2 すべての Thumb-2 ISA

MCR2<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}




```
t = UInt(Rt);  cp = UInt(coproc);
if t == 15 || t == 13 then UNPREDICTABLE;
```

アセンブラ構文

```
MCR{2}<c><q>  <coproc>, #<opc1>, <Rt>, <CRn>, <CRm>{, #<opc2>}
```

各項目の説明については以下を参照して下さい。

2 指定した場合、エンコードの C == 1 の形式が選択されます。省略した場合、C == 0 の形式が選択されます。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<coproc> コプロセッサの名前を指定します。標準の汎用コプロセッサ名は、p0、p1、...、p15 です。

<opc1> コプロセッサ固有のオペコードで、範囲は 0 ～ 7 です。

<Rt> 値をコプロセッサに転送する ARM レジスタ

<CRn> デスティネーション コプロセッサレジスタ

<CRm> 追加のデスティネーション コプロセッサレジスタ

<opc2> コプロセッサ固有のオペコードで、範囲は 0 ～ 7 です。省略した場合、<opc2> が 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_SendOneWord(R[t], cp, ThisInstr());
```

例外

UsageFault

注

コプロセッサのフィールド

ARM アーキテクチャでは、命令のビット <31:24>、<20>、<15:8>、<4> のみが定義されています。他のフィールドは推奨事項です。

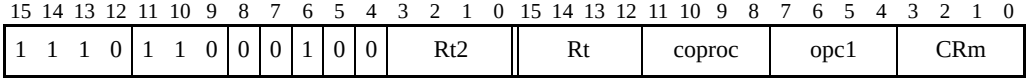
A6.7.72 MCRR, MCRR2

MCRR、MCRR2（2 つの ARM レジスタからコプロセッサへの移動）は、2 つの ARM レジスタの値をコプロセッサに転送します。

コプロセッサが命令を実行できない場合は、UsageFault 例外が発生します。

エンコード T1 すべての Thumb-2 ISA

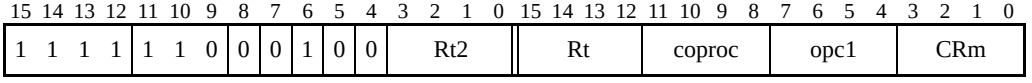
MCRR<c> <coproc>,<opc1>,<Rt>,<Rt2>,<CRm>




```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
```

エンコード T2 すべての Thumb-2 ISA

MCRR2<c> <coproc>,<opc1>,<Rt>,<Rt2>,<CRm>




```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
```

アセンブラ構文

```
MCRR{2}<c><q>  <coproc>, #<opc1>, <Rt>, <Rt2>, <CRm>
```

各項目の説明については以下を参照して下さい。

2　指定した場合、エンコードの C == 1 の形式が選択されます。省略した場合、C == 0 の形式が選択されます。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<coproc>　コプロセッサの名前を指定します。
標準の汎用コプロセッサ名は、p0、p1、...、p15 です。

<opc1>　コプロセッサ固有のオペコードで、範囲は 0 ～ 15 です。

<Rt>　値をコプロセッサに転送する最初の ARM レジスタ

<Rt2>　値をコプロセッサに転送する 2 番目の ARM レジスタ

<CRm>　デスティネーション コプロセッサレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_SendTwoWords(R[t], R[t2], cp, ThisInstr());
```

例外

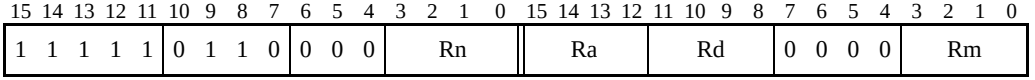
UsageFault

A6.7.73 MLA

MLA（積和演算）は、2 つのレジスタの値を乗算して、3 番目のレジスタの値を加算します。結果の下位 32 ビットがデスティネーションレジスタに書き込まれます。この 32 ビットは、符号付き計算と符号なし計算のどちらが実行されたかに依存しません。

エンコード T1　　ARMv6T2、ARMv7

MLA<c> <Rd>,<Rn>,<Rm>,<Ra>




```
if Ra == '1111' then SEE MUL;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = FALSE;
if BadReg(d) || BadReg(n) || BadReg(m) || a == 13 then UNPREDICTABLE;
```

アセンブラ構文

```
MLA<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　2 番目のオペランドを格納しているレジスタを指定します。

<Ra>　　加算される値を格納しているレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = operand1 * operand2 + addend;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        // APSR.C unchanged
        // APSR.V unchanged
```

例外

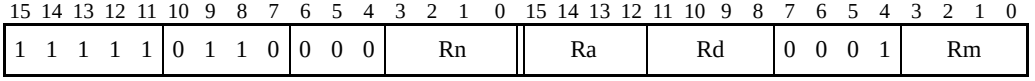
なし

A6.7.74 MLS

MLS（積減算）は、2 つのレジスタ値を乗算して、結果の下位 32 ビットを 3 番目のレジスタから減算します。この 32 ビットは、符号付き計算と符号なし計算のどちらが実行されたかに依存しません。結果はデスティネーションレジスタに書き込まれます。

エンコード T1　　ARMv6T2、ARMv7

MLS<c> <Rd>,<Rn>,<Rm>,<Ra>




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if BadReg(d) || BadReg(n) || BadReg(m) || BadReg(a) then UNPREDICTABLE;
```

アセンブラ構文

```
MLS<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　2 番目のオペランドを格納しているレジスタを指定します。

<Ra>　　減算される値を格納しているレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = addend - operand1 * operand2;
    R[d] = result<31:0>;
```

例外

なし

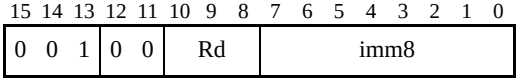
A6.7.75 MOV（イミディエート）

MOV（移動（イミディエート））は、イミディエート値をデスティネーションレジスタに書き込みます。オプションとして、その値に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

MOVS <Rd>,#<imm8> IT ブロック外部

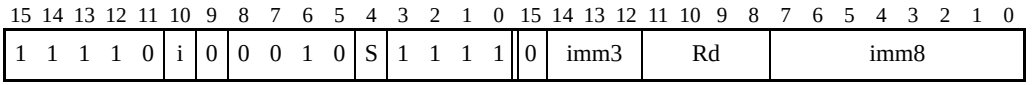
MOV<c> <Rd>,#<imm8> IT ブロック内部




```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

エンコード T2 ARMv6T2、ARMv7

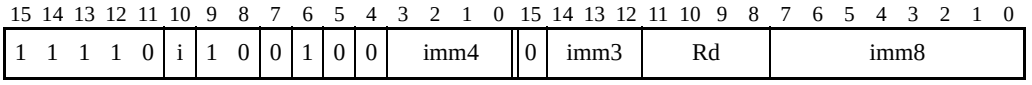
MOV{S}<c>.W <Rd>,#<const>




```
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

MOVW<c> <Rd>,#<imm16>




```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
if BadReg(d) then UNPREDICTABLE;
```


アセンブラ構文

MOV{S}<c><q> <Rd>, #<const>	すべてのエンコードが許可される
MOVW<c><q> <Rd>, #<const>	エンコード T3 のみ許可される

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。

<const>　<Rd> に書き込まれるイミディエート値を指定します。許容される値の範囲は、エンコード T1 では 0 ～ 255、エンコード T3 では 0 ～ 65535 です。エンコード T2 で許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

両方の 32 ビットエンコードが使用可能な場合は、エンコード T3 よりもエンコード T2 の方が適しています (エンコード T3 が必要な場合は、MOVW 構文を使用して下さい)。

UAL 以前の構文の MOV<c>S は、MOVS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = imm32;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

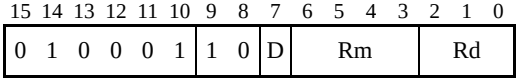
A6.7.76 MOV（レジスタ）

MOV（移動（レジスタ））は、レジスタからデスティネーションレジスタに値をコピーします。オプションとして、その値に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6*、ARMv7（<Rd> と <Rm> の R0 ～ R7 の場合）
ARMv4T、ARMv5T*、ARMv6*、ARMv7（上記以外の場合）

MOV<c> <Rd>,<Rm>

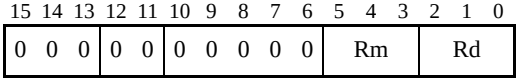
<Rd> が PC の場合は、IT ブロックの外部に配置するか、IT ブロック内部の最後の命令にする必要がある。




```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

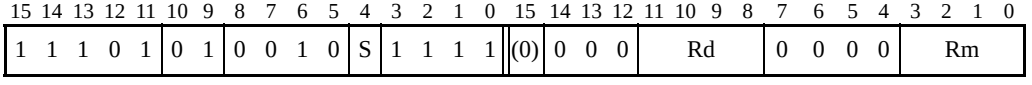
MOVS <Rd>,<Rm>　（以前は、LSL <Rd>,<Rm>,#0）　IT ブロック内部では許可されない




```
d = UInt(Rd);  m = UInt(Rm);  setflags = TRUE;
if InITBlock() then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

MOV{S}<c>.W <Rd>,<Rm>




```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
if (d == 13 || BadReg(m)) && setflags then UNPREDICTABLE;
if d == 13 && BadReg(m) then UNPREDICTABLE;
if d == 15 then UNPREDICTABLE;
```

アセンブラ構文

```
MOV{S}<c><q>  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタ。S を指定しない場合、このレジスタに SP または PC を指定できます。

<Rd> が PC の場合は、エンコード T1 のみが許可され、命令によって PC に移動されたアドレスへの分岐が発生します。この命令は、IT ブロック外部に配置するか、IT ブロック内部の最後の命令にする必要があります。

<Rd> が SP で <Rm> が SP または PC の場合は、エンコード T3 は許可されません。

<Rm>　　ソースレジスタ。S を指定しない場合、このレジスタに SP または PC を指定できます。

注

<Rd> が SP または PC で、<Rm> も SP または PC である MOV（レジスタ）命令の使用は推奨されません。

UAL 以前の構文の MOV<c>S は、MOVS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[m];
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

例外

なし

A6.7.77 MOV（シフトしたレジスタ）

MOV（移動（シフトしたレジスタ））は、ASR、LSL、LSR、ROR、およびRRXと同義語です。

詳細については、次のセクションを参照して下さい。

- *ASR（イミディエート）*: P.A6-36
- *ASR（レジスタ）*: P.A6-38
- *LSL（イミディエート）*: P.A6-135
- *LSL（レジスタ）*: P.A6-137
- *LSR（イミディエート）*: P.A6-139
- *LSR（レジスタ）*: P.A6-141
- *ROR（イミディエート）*: P.A6-193
- *ROR（レジスタ）*: P.A6-195
- *RRX*: P.A6-197

アセンブラ構文

MOV（シフトしたレジスタ）と他の命令の対応を、表 A6-4 に示します。

表 A6-4 MOV（シフト、レジスタシフト）と等価な命令

MOV 命令	標準形式
MOV{S} <Rd>,<Rm>,ASR #<n>	ASR{S} <Rd>,<Rm>,#<n>
MOV{S} <Rd>,<Rm>,LSL #<n>	LSL{S} <Rd>,<Rm>,#<n>
MOV{S} <Rd>,<Rm>,LSR #<n>	LSR{S} <Rd>,<Rm>,#<n>
MOV{S} <Rd>,<Rm>,ROR #<n>	ROR{S} <Rd>,<Rm>,#<n>
MOV{S} <Rd>,<Rm>,ASR <Rs>	ASR{S} <Rd>,<Rm>,<Rs>
MOV{S} <Rd>,<Rm>,LSL <Rs>	LSL{S} <Rd>,<Rm>,<Rs>
MOV{S} <Rd>,<Rm>,LSR <Rs>	LSR{S} <Rd>,<Rm>,<Rs>
MOV{S} <Rd>,<Rm>,ROR <Rs>	ROR{S} <Rd>,<Rm>,<Rs>
MOV{S} <Rd>,<Rm>,RRX	RRX{S} <Rd>,<Rm>

命令の標準形式は、逆アセンブルで生成されます。

例外

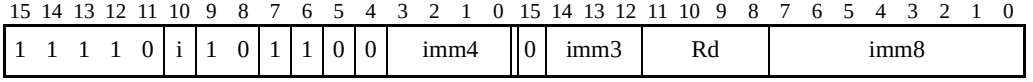
なし

A6.7.78 MOVT

MOVT（上位ビットの移動）は、イミディエート値をデスティネーションレジスタの上位ハーフワードに書き込みます。下位ハーフワードの内容には影響を与えません。

エンコード T1　　ARMv6T2、ARMv7

```
MOVT<c> <Rd>,#<imm16>
```




```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if BadReg(d) then UNPREDICTABLE;
```

アセンブラ構文

```
MOVT<c><q>  <Rd>, #<imm16>
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<imm16>　　<Rd> に書き込むイミディエート値を指定します。許容される値の範囲は 0 〜 65535 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<31:16> = imm16;
    // R[d]<15:0> unchanged
```

例外

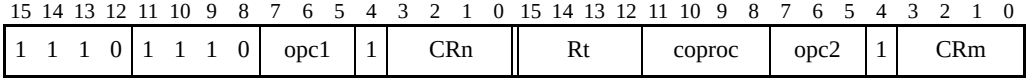
なし

A6.7.79 MRC, MRC2

MRC、MRC2（コプロセッサから ARM レジスタへの移動）は、コプロセッサに対して、ARM レジスタまたは条件フラグへ値を転送するように指示します。

エンコード T1 すべての Thumb-2 ISA

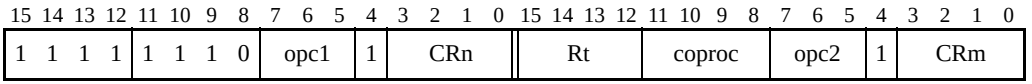
MRC<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}




```
t = UInt(Rt);  cp = UInt(coproc);
if t == 13 then UNPREDICTABLE;
```

エンコード T2 すべての Thumb-2 ISA

MRC2<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}




```
t = UInt(Rt);  cp = UInt(coproc);
if t == 13 then UNPREDICTABLE;
```

コプロセッサが命令を実行できない場合は、UsageFault 例外が発生します。

アセンブラ構文

```
MRC{2}<c><q>  <coproc>, #<opc1>, <Rt>, <CRn>, <CRm>{, #<opc2>}
```

各項目の説明については以下を参照して下さい。

2　指定した場合、エンコードの C == 1 の形式が選択されます。省略した場合、C == 0 の形式が選択されます。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<coproc>　コプロセッサの名前を指定します。標準の汎用コプロセッサ名は、p0、p1、...、p15 です。

<opc1>　コプロセッサ固有のオペコードで、範囲は 0 ～ 7 です。

<Rt>　デスティネーションの ARM レジスタ。R0 ～ R14 または APSR_nzcv を指定できます。最後の形式は、転送される値のビット <31:28> を、N、Z、C、V の条件フラグに書き込みます。これは、エンコードの Rt フィールドを 0b1111 に設定することによって指定されます。UAL 以前のアセンブラ構文では、この形式を選択するために APSR_nzcv の代わりに PC が書き込まれます。

<CRn>　最初のオペランドを含むコプロセッサレジスタ

<CRm>　追加のソースまたはデスティネーション コプロセッサレジスタ

<opc2>　コプロセッサ固有のオペコードで、範囲は 0 ～ 7 です。省略した場合、<opc2> が 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        value = Coproc_GetOneWord(cp, ThisInstr());
        if t != 15 then
            R[t] = value;
        else
            APSR.N = value<31>;
            APSR.Z = value<30>;
            APSR.C = value<29>;
            APSR.V = value<28>;
            // value<27:0> are not used.
```

例外

UsageFault

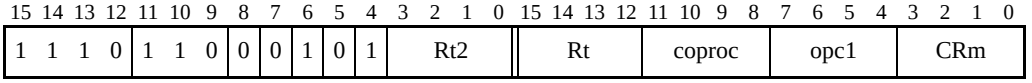
A6.7.80 MRRC、MRRC2

MRRC、MRRC2（コプロセッサから 2 つの ARM レジスタへの移動）は、コプロセッサに対して、2 つの ARM レジスタへ値を転送するように指示します。

コプロセッサが命令を実行できない場合は、UsageFault 例外が発生します。

エンコード T1 すべての Thumb-2 ISA

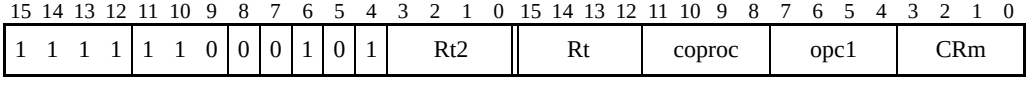
MRRC<c> <coproc>,<opc>,<Rt>,<Rt2>,<CRm>




```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
```

エンコード T2 すべての Thumb-2 ISA

MRRC2<c> <coproc>,<opc>,<Rt>,<Rt2>,<CRm>




```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
```

アセンブラ構文

```
MRRC{2}<c><q>  <coproc>, #<opc1>, <Rt>, <Rt2>, <CRm>
```

各項目の説明については以下を参照して下さい。

2　指定した場合、エンコードの C == 1 の形式が選択されます。省略した場合、C == 0 の形式が選択されます。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<coproc>　コプロセッサの名前を指定します。標準の汎用コプロセッサ名は、p0、p1、...、p15 です。

<opc1>　コプロセッサ固有のオペコードで、範囲は 0 ～ 15 です。

<Rt>　最初のデスティネーション ARM レジスタです。

<Rt2>　2 番目のデスティネーション ARM レジスタです。

<CRm>　転送されるデータを供給するコプロセッサレジスタ

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        (R[t], R[t2]) = Coproc_GetTwoWords(cp, ThisInstr());
```

例外

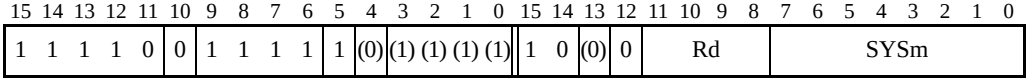
UsageFault

A6.7.81 MRS

MRS（特殊レジスタからレジスタへの移動）は、選択された専用レジスタから汎用 ARM レジスタへ値を移動します。

エンコード T1 ARMv6T2、ARMv7

MRS<c> <Rd>,<spec_reg>



注

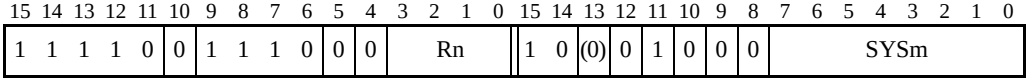
MRS は、APSR レジスタ (SYSm = 0) または CONTROL レジスタ (SYSm = 0x14) にアクセスするとき以外は、システムレベル命令です。命令の完全な定義については、P.B3-4「*MRS*」を参照して下さい。

A6.7.82 MSR（レジスタ）

MSR（ARM レジスタから特殊レジスタへの移動）は、汎用 ARM レジスタの値を、指定された専用レジスタへ移動します。

エンコード T1 ARMv6T2、ARMv7

MSR<c> <spec_reg>,<Rn>



注

MSR（レジスタ）は、APSR レジスタ (SYSm = 0) にアクセスするとき以外は、システムレベル命令です。命令の完全な定義については、P.B3-8「*MSR（レジスタ）*」を参照して下さい。

A6.7.83 MUL

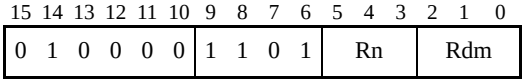
MUL（乗算）は、2 つのレジスタの値を乗算します。結果の下位 32 ビットがデスティネーションレジスタに書き込まれます。この 32 ビットは、符号付き計算と符号なし計算のどちらが実行されたかに依存しません。

オプションとして、結果に基づいて条件フラグを更新することができます。このオプションは、Thumb 命令セット内の命令のいくつかの形式に限られており、使用した場合に、多くのプロセッサ実装で性能に悪影響を及ぼします。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

MULS <Rdm>,<Rn>,<Rdm> IT ブロック外部

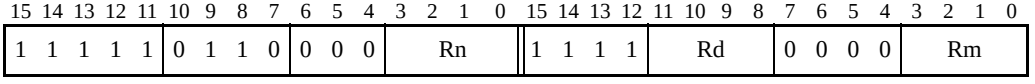
MUL<c> <Rdm>,<Rn>,<Rdm> IT ブロック内部




```
d = UInt(Rdm);  n = UInt(Rn);  m = UInt(Rdm);  setflags = !InITBlock();
```

エンコード T2 ARMv6T2、ARMv7

MUL<c> <Rd>,<Rn>,<Rm>




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
MUL{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> と同じレジスタが使用されます。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　2 番目のオペランドを含むレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        // APSR.C unchanged
        // APSR.V unchanged
```

例外

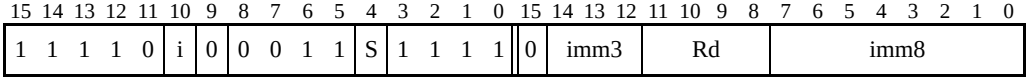
なし

A6.7.84 MVN（イミディエート）

MVN（ビット単位 NOT（イミディエート））は、イミディエート値のビット単位の否定をデスティネーションレジスタに書き込みます。オプションとして、その値に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

MVN{S}<c> <Rd>,#<const>




```
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) then UNPREDICTABLE;
```

アセンブラ構文

```
MVN{S}<c><q>  <Rd>, #<const>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<const>　　<Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

UAL 以前の構文の MVN<c>S は、MVNS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = NOT(imm32);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

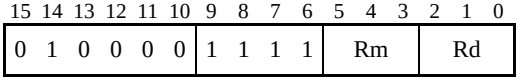
A6.7.85 MVN（レジスタ）

MVN（ビット単位否定（レジスタ））は、レジスタの値のビット単位 NOT をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

MVNS <Rd>,<Rm> IT ブロック外部

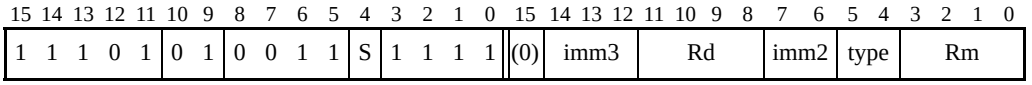
MVN<c> <Rd>,<Rm> IT ブロック内部




```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

MVN{S}<c>.W <Rd>,<Rm>{,shift>}




```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
MVN{S}<c><q>  <Rd>, <Rm> {, <shift>}
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rm>　　オプションとしてシフトされ、ソースレジスタとして使用されるレジスタを指定します。

<shift>　　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われず、両方のエンコードが許可されます。<shift> を指定した場合、エンコード T2 のみが許可されます。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

UAL 以前の構文の MVN<c>S は、MVNS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

A6.7.86 NEG

NEG（否定）は、UAL 以前に使用されていた同義語で、RSB（イミディエート）においてイミディエート値が 0 の場合と等価です。詳細については、P.A6-199「*RSB（イミディエート）*」を参照して下さい。

アセンブラ構文

```
NEG<c><q>  {<Rd>,} <Rm>
```

これは、次の命令と等価です。

```
RSBS<c><q>  {<Rd>,} <Rm>, #0
```

例外

なし

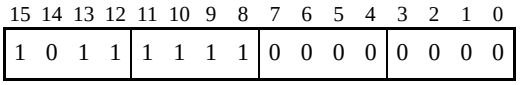
A6.7.87 NOP

NOP（無操作）は何の動作も行いません。

この命令は、NOP 互換ヒント（アーキテクチャ上の NOP）です。詳細については、P.A6-16「*NOP 互換ヒント*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

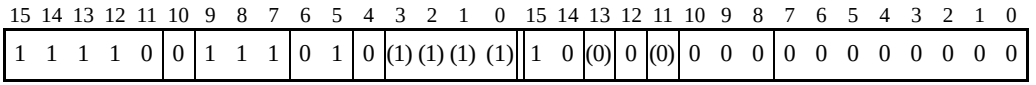
NOP<c>




```
// No additional decoding required
```

エンコード T2 ARMv6T2、ARMv7

NOP<c>.W




```
// No additional decoding required
```

アセンブラ構文

```
NOP<c><q>
```

各項目の説明については以下を参照して下さい。

〈c〉〈q〉 P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
```

例外

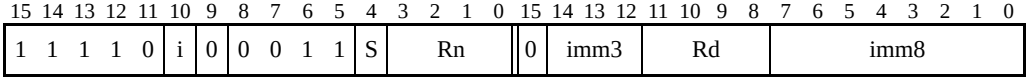
なし

A6.7.88 ORN（イミディエート）

ORN（否定論理和（イミディエート））は、レジスタの値と、イミディエート値の補数とのビット単位の（非排他的）論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

ORN{S}<c> <Rd>,<Rn>,#<const>




```
if Rn == '1111' then SEE MVN (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || n == 13 then UNPREDICTABLE;
```

アセンブラ構文

```
ORN{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> と同じレジスタが使用されます。

<Rn>　オペランドを含むレジスタを指定します。

<const>　<Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR NOT(imm32);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

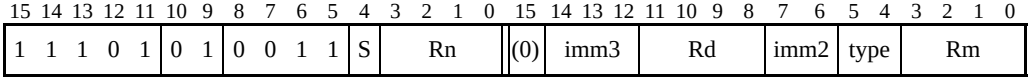
なし

A6.7.89 ORN（レジスタ）

ORN（否定論理和（レジスタ））は、レジスタの値と、オプションとしてシフトしたレジスタの値の補数とでビット単位（非排他的）論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

ORN{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}




```
if Rn == '1111' then SEE MVN (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
ORN{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> と同じレジスタが使用されます。

<Rn>　第一オペランドを格納しているレジスタを指定します。

<Rm>　オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタを指定します。

<shift>　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われません。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

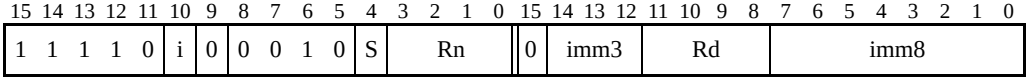
なし

A6.7.90 ORR（イミディエート）

ORR（論理和（イミディエート））は、レジスタの値とイミディエート値とのビット単位（非排他的）の論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2, ARMv7

ORR{S}<c> <Rd>,<Rn>,#<const>




```
if Rn == '1111' then SEE MOV (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(d) || n == 13 then UNPREDICTABLE;
```

アセンブラ構文

```
ORR{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

<Rn>　　オペランドを含むレジスタを指定します。

<const>　　<Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

UAL 以前の構文の ORR<c>S は、ORRS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR imm32;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

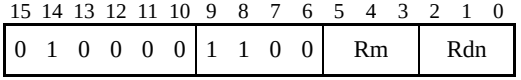
A6.7.91 ORR（レジスタ）

ORR（論理和（レジスタ））は、レジスタの値と、オプションとしてシフトされたレジスタの値とのビット単位（非排他的）の論理和を実行して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

ORRS <Rdn>,<Rm> IT ブロック外部

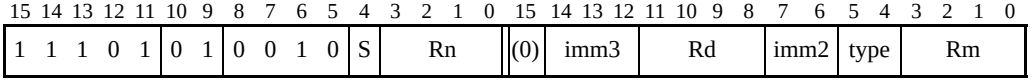
ORR<c> <Rdn>,<Rm> IT ブロック内部




```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

ORR{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}




```
if Rn == '1111' then SEE MOV (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || n == 13 || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
ORR{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合は、<Rn> が使用されます。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタを指定します。

<shift>　　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われず、両方のエンコードが許可されます。<shift> を指定した場合、エンコード T2 のみが許可されます。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

特殊なケースとして、ORR<c> <Rd>,<Rn>,<Rd> と記述し、<Rd> および <Rn> がいずれも R0 ～ R7 である場合、ORR<c> <Rd>,<Rn> と記述されたものとして、エンコード T2 を使用してアセンブルされます。これを避けるには、.W 修飾子を使用します。

UAL 以前の構文の ORR<c>S は、ORRS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

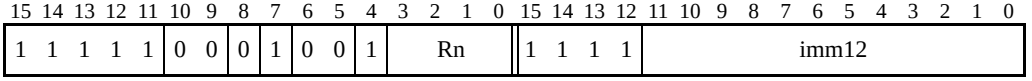
なし

A6.7.92 PLD（イミディエート、リテラル）

PLD（データのプリロード（イミディエート、リテラル））は、指定されたアドレスからのデータメモリ アクセスが近く発生する可能性があることをメモリシステムに伝達することができるメモリヒント命令です。メモリシステムは、ヒントへの応答として、指定されたアドレスを含むキャッシュラインをデータキャッシュにプリロードするなど、メモリアクセスを高速化する動作を実行できます。詳細については、P.A3-40「*キャッシュのプリロード*」および P.A6-16「*メモリヒント*」を参照して下さい。

エンコード T1　　ARMv6T2, ARMv7

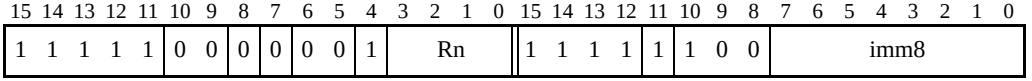
PLD<c> [<Rn>,#<imm12>]




```
if Rn == '1111' then SEE encoding T3;
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;  is_pldw = (W == '1');
```

エンコード T2　　ARMv6T2, ARMv7

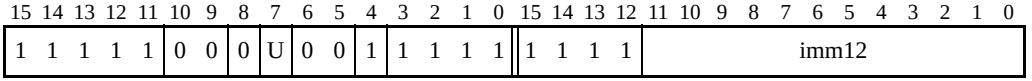
PLD<c> [<Rn>,#-<imm8>]




```
if Rn == '1111' then SEE encoding T3;
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;  is_pldw = (W == '1');
```

エンコード T3　　ARMv6T2, ARMv7

PLD<c> <label>




```
n = 15;  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');  is_pldw = (W == '1');
```

アセンブラ構文

```
PLD<c><q>  [<Rn>, #+/-<imm>]
PLD<c><q>  [PC, #+/-<imm>]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　ベースレジスタ。SP を指定できます。

+/-　イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm>　ベースレジスタからのオフセットを指定します。許容される値の範囲は次のとおりです。

- -4095 ～ 4095（ベースレジスタが PC の場合）
- -255 ～ 4095（上記以外の場合）

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    Hint_PreloadData(address);
```

例外

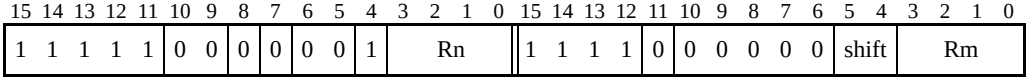
なし

A6.7.93 PLD（レジスタ）

PLD（データのプリロード（レジスタ））は、指定されたアドレスからのデータメモリ アクセスが近く発生する可能性があることをメモリシステムに伝達することができるメモリヒント命令です。メモリシステムは、ヒントへの応答として、指定されたアドレスを含むキャッシュラインをデータキャッシュにプリロードするなど、メモリアクセスを高速化する動作を実行できます。詳細については、P.A3-40「*キャッシュのプリロード*」および P.A6-16「*メモリヒント*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

PLD<c> [<Rn>,<Rm>{,LSL #<imm2>}]




```
if Rn == '1111' then SEE PLD (immediate);
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;  is_pldw = (W == '1');
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
PLD<c><q>  [<Rn>, <Rm> {, LSL #<shift>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　ベースレジスタ。SP を指定できます。

<Rm>　オプションでシフトされるオフセットレジスタ

<shift>　<Rm> から読み出した値に適用されるシフトを、0 ～ 3 の範囲で指定します。このオプションを省略した場合、0 ビットのシフトと見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadData(address);
```

例外

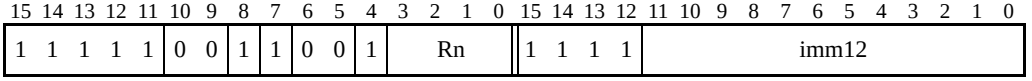
なし

A6.7.94 PLI（イミディエート、リテラル）

PLI（命令のプリロード（イミディエート、リテラル））は、指定されたアドレスからの命令メモリアクセスが近く発生する可能性があることをメモリシステムに伝達することができるメモリヒント命令です。メモリシステムは、ヒントへの応答として、指定されたアドレスを含むキャッシュラインを命令キャッシュにプリロードするなど、メモリアクセスを高速化するための動作を実行できます。詳細については、P.A3-40「*キャッシュのプリロード*」、P.A3-40、P.A3-40「*キャッシュのプリロード*」および P.A6-16「*メモリヒント*」を参照して下さい。

エンコード T1　　ARMv7

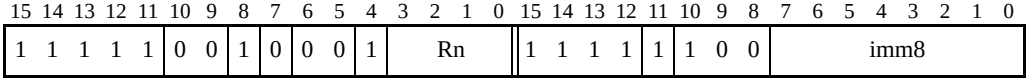
PLI<c> [<Rn>,#<imm12>]




```
if Rn == '1111' then SEE encoding T3;
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;
```

エンコード T2　　ARMv7

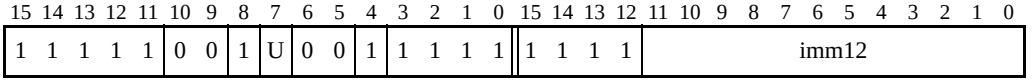
PLI<c> [<Rn>,#-<imm8>]




```
if Rn == '1111' then SEE encoding T3;
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;
```

エンコード T3　　ARMv7

PLI<c> <label>




```
n = 15;  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

アセンブラ構文

```
PLI<c><q>  [<Rn>, #+/-<imm>]
PLI<c><q>  [PC, #+/-<imm>]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　ベースレジスタ。SP を指定できます。

+/-　イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm>　ベースレジスタからのオフセットを指定します。許容される値の範囲は次のとおりです。

- -4095 ～ 4095（ベースレジスタが PC の場合）
- -255 ～ 4095（上記以外の場合）

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
```

例外

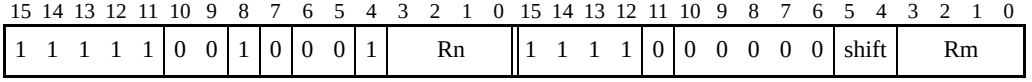
なし

A6.7.95 PLI（レジスタ）

PLI（命令のプリロード（レジスタ））は、指定されたアドレスからの命令メモリアクセスが近く発生する可能性があることをメモリシステムに伝達することができるメモリヒント命令です。メモリシステムは、ヒントへの応答として、指定されたアドレスを含むキャッシュラインを命令キャッシュにプリロードするなど、メモリアクセスを高速化する動作を実行できます。詳細については、P.A3-40「*キャッシュのプリロード*」およびP.A6-16「*メモリヒント*」を参照して下さい。

エンコードT1 ARMv7

PLI<c> [<Rn>,<Rm>{,LSL #<imm2>}]




```
if Rn == '1111' then SEE PLI (immediate);
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
PLI<c><q>  [<Rn>, <Rm> {, LSL #<shift>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　　ベースレジスタ。SP を指定できます。

<Rm>　　オプションとしてシフトされる、オフセットレジスタ

<shift>　　<Rm> から読み出した値に適用するシフトを、0 ～ 3 の範囲で指定します。このオプションを省略した場合、0 ビットのシフトと見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadInstr(address);
```

例外

なし

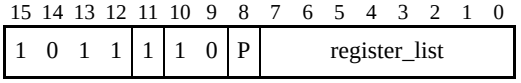
A6.7.96 POP

POP（複数レジスタポップ）は、汎用レジスタ R0 ～ R12 のサブセット（または全部）と PC または LR をスタックからロードします。

ロードするレジスタに PC が含まれている場合、PC にロードされるワードはアドレスまたは例外の戻り値として扱われ、分岐が発生します。ビット <0> は、Thumb 状態実行への分岐に関する ARM アーキテクチャのインターワーキングルールに準拠するために、1 に設定する必要があります。ビット <0> が 0 の場合は、UsageFault 例外が発生します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

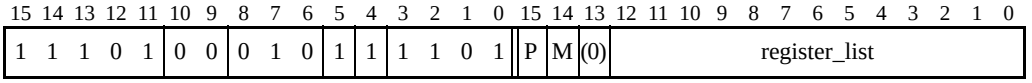
POP<c> <registers>




```
registers = P:'0000000':register_list;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

エンコード T2 ARMv6T2, ARMv7

POP<c>.W <registers>




```
registers = P:M:'0':register_list;
if BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

アセンブラ構文

POP<c><q> <registers>	標準構文
LDMIA<c><q> SP!, <registers>	等価な LDM 構文

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<registers>

1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストで、ロードされるレジスタのセットを指定します。最下位のメモリアドレスにあるデータは最も小さい番号のレジスタにロードされ、以下順にロードが行われ、最上位のメモリアドレスにあるデータは最も大きい番号のレジスタにロードされます。レジスタリストに PC を指定した場合は、この命令によって、PC にロードされたアドレス（データ）への分岐が発生します。

エンコード T2 では、1 つのレジスタしか含まないリストはサポートされません。リストに <Rt> しか含まない POP 命令が Thumb にアセンブルされ、エンコード T1 が使用できない場合は、それと等価な LDR<c><q> <Rt>,[SP],#-4 命令にアセンブルされます。

リストに SP を含めることはできません。

リストに PC が存在する場合、LR をリストに含めることはできません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP;

    for i = 0 to 14
        if registers<i> == '1' then
            R[i} = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);
```

例外

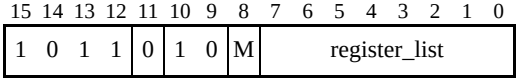
UsageFault、MemManage、BusFault

A6.7.97 PUSH

PUSH（複数レジスタプッシュ）は、汎用レジスタ R0 ～ R12 のサブセット（または全部）と LR をスタックにストアします。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

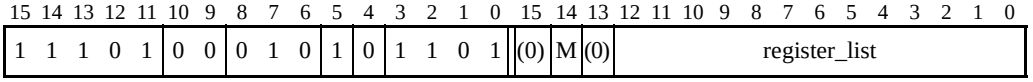
PUSH<c> <registers>




```
registers = '0':M:'000000':register_list;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

PUSH<c>.W <registers>




```
registers = '0':M:'0':register_list;
if BitCount(registers) < 2 then UNPREDICTABLE;
```

アセンブラ構文

PUSH<c><q> <registers>	標準構文
STMDB<c><q> SP!, <registers>	等価な STM 構文

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<registers>

1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストで、ストアされるレジスタのセットを指定します。最も小さい番号のレジスタが最下位のメモリアドレスにストアされ、以下順にストアが行われ、最も大きい番号のレジスタが最上位のメモリアドレスにストアされます。

T2 エンコードでは、1 つのレジスタしか含まないリストはサポートされません。リストに <Rt> しか含まない PUSH 命令が Thumb にアセンブルされ、エンコード T1 が使用できない場合は、等価な STR<c><q> <Rt>,[SP,#-4]! 命令にアセンブルされます。

リストに SP と PC を含めることはできません。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP - 4*BitCount(registers);

    for i = 0 to 14
        if registers<i> == '1' then
            MemA[address,4] = R[i];
            address = address + 4;

    SP = SP - 4*BitCount(registers);
```

例外

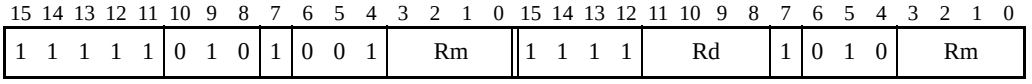
UsageFault、MemManage、BusFault

A6.7.98 RBIT

RBIT（ビット反転）は、32 ビットレジスタのビット順序を反転します。

エンコード T1　　ARMv6T2、ARMv7

RBIT<c> <Rd>,<Rm>




```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

RBIT<c><q> <Rd>, <Rm>

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rm>　　オペランドを含むレジスタを指定します。エンコード T1 では、Rm フィールドと Rm2 フィールドの両方で、その番号を 2 回エンコードする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    for i = 0 to 31 do
        result<31-i> = R[m]<i>;
    R[d] = result;
```

例外

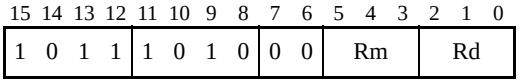
なし

A6.7.99 REV

REV（ワードのバイト反転）は、32 ビットレジスタのバイト順序を反転します。

エンコード T1 ARMv6 またはそれ以降のすべてのバージョンの Thumb ISA

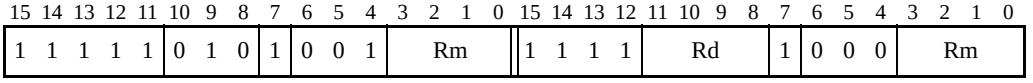
```
REV<c> <Rd>,<Rm>
```




```
d = UInt(Rd);  m = UInt(Rm);
```

エンコード T2 ARMv6T2、ARMv7

```
REV<c>.W <Rd>,<Rm>
```




```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
REV<c><q>  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<Rm> オペランドを含むレジスタを指定します。T2 エンコードでは、Rm フィールドと Rm2 フィールドの両方で、その番号を 2 回エンコードする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<7:0>;
    result<23:16> = R[m]<15:8>;
    result<15:8>  = R[m]<23:16>;
    result<7:0>   = R[m]<31:24>;
    R[d] = result;
```

例外

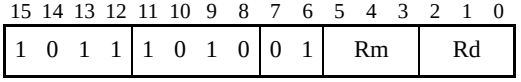
なし

A6.7.100 REV16

REV16（パックハーフワードのバイト反転）は、32 ビットレジスタの各 16 ビットハーフワードのバイト順序を反転します。

エンコード T1　　ARMv6 またはそれ以降のすべてのバージョンの Thumb ISA

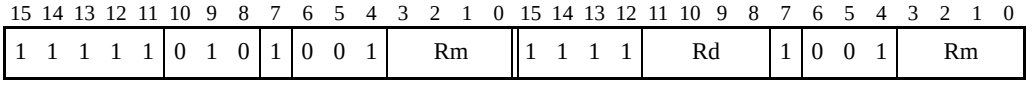
```
REV16<c> <Rd>,<Rm>
```




```
d = UInt(Rd);  m = UInt(Rm);
```

エンコード T2　　ARMv6T2、ARMv7

```
REV16<c>.W <Rd>,<Rm>
```




```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
REV16<c><q>  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rm>　　オペランドを含むレジスタを指定します。T2 エンコードでは、Rm フィールドと Rm2 フィールドの両方で、その番号を 2 回エンコードする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<23:16>;
    result<23:16> = R[m]<31:24>;
    result<15:8>  = R[m]<7:0>;
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

例外

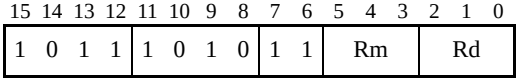
なし

A6.7.101 REVSH

REVSH（符号付きハーフワードのバイト反転）は、32 ビットレジスタの下位 16 ビットハーフワードのバイト順序を反転して、結果を 32 ビットに符号拡張します。

エンコード T1 ARMv6 またはそれ以降のすべてのバージョンの Thumb ISA

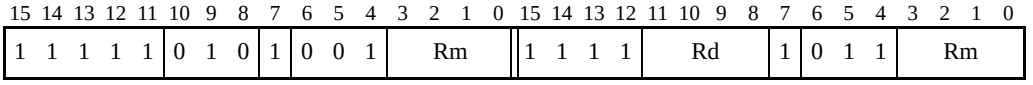
REVSH<c> <Rd>,<Rm>




```
d = UInt(Rd);  m = UInt(Rm);
```

エンコード T2 ARMv6T2、ARMv7

REVSH<c>.W <Rd>,<Rm>




```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
REVSH<c><q>  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<Rm> オペランドを含むレジスタを指定します。エンコード T2 では、Rm フィールドと Rm2 フィールドの両方で、その番号を 2 回エンコードする必要があります。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:8>  = SignExtend(R[m]<7:0>, 24);
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

例外

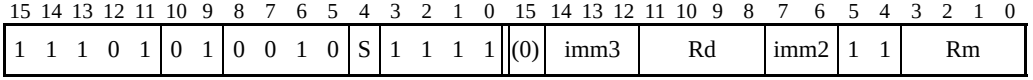
なし

A6.7.102 ROR（イミディエート）

ROR（右ローテート（イミディエート））は、定数値で示されるビット数だけレジスタの内容をローテートします。ローテートによって右端からはみ出したビットは、左側の空きビットの位置に挿入されます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1　　ARMv6T2、ARMv7

ROR{S}<c> <Rd>,<Rm>,#<imm5>




```
if (imm3:imm2) == '00000' then SEE RRX;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('11', imm3:imm2);
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
ROR{S}<c><q>  <Rd>, <Rm>, #<imm5>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rm>　　第一オペランドを格納しているレジスタを指定します。

<imm5>　　シフトするビット数を 1 ～ 31 の範囲で指定します。詳細については、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_ROR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

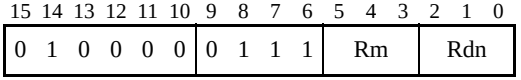
A6.7.103 ROR（レジスタ）

ROR（右ローテート（レジスタ））は、可変のビット数分だけレジスタの内容をローテートします。ローテートによって右端からはみ出したビットは、左側の空きビットの位置に挿入されます。ローテートするビット数は、レジスタの下位バイトから読み出されます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

RORS <Rdn>,<Rm> IT ブロック外部

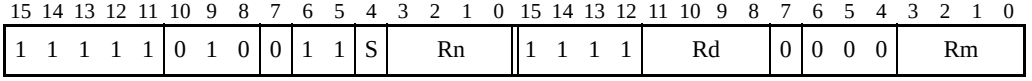
ROR<c> <Rdn>,<Rm> IT ブロック内部




```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

エンコード T2 ARMv6T2, ARMv7

ROR{S}<c>.W <Rd>,<Rn>,<Rm>




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
ROR{S}<c><q>  <Rd>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　下位バイトにローテートのビット数を含むレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_ROR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

A6.7.104 RRX

RRX（拡張付き右ローテート）は、レジスタの内容を 1 ビット分右にシフトします。ビット <31> にはキャリーフラグがシフトされます。

RRX オプションとして、結果に基づいて条件フラグを更新することができます。この場合、ビット <0> がキャリーフラグにシフトされます。

エンコード T1 ARMv6T2、ARMv7

RRX{S}<c> <Rd>,<Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	0	0	0	Rd				0	0	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
RRX{S}<c><q>  <Rd>, <Rm>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rm>　　オペランドを含むレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_RRX, 1, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

例外

なし

A6.7.105 RSB（イミディエート）

RSB 令（逆減算（イミディエート））は、イミディエート値からレジスタの値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1　　ARMv4T、ARMv5T*、ARMv6*、ARMv7

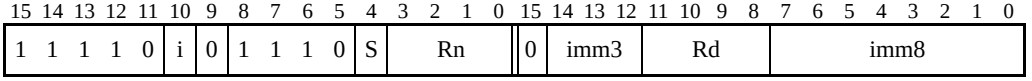
RSBS <Rd>,<Rn>,#0　　IT ブロック外部

RSB<c> <Rd>,<Rn>,#0　　IT ブロック内部


```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = Zeros(32); // immediate = #0
```

エンコード T2　　ARMv6T2、ARMv7

RSB{S}<c>.W <Rd>,<Rn>,#<const>




```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

アセンブラ構文

```
RSB{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<const>　　<Rn> から取得された値に加算されるイミディエート値を指定します。エンコード T1 で許容される値は 0 のみです。エンコード T2 で許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

UAL 以前の構文の RSB<c>S は、RSBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), imm32, '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

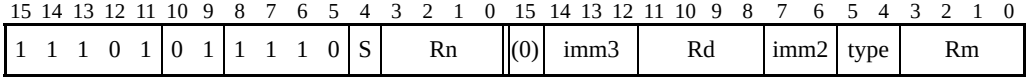
なし

A6.7.106 RSB（レジスタ）

RSB（逆減算（レジスタ））は、オプションでシフトしたレジスタの値からレジスタの値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1　　ARMv6T2、ARMv7

RSB{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
RSB{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が指定されます。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　オプションでシフトされ、2 番目のオペランドとして使用されるレジスタを指定します。

<shift>　　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われません。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

UAL 以前の構文の RSB<c>S は、RSBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), shifted, ‘1’);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.C = carry;
        APSR.V = overflow;
```

例外

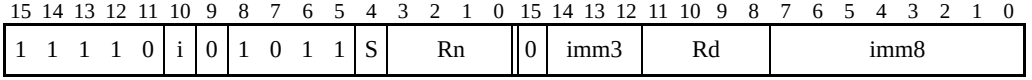
なし

A6.7.107 SBC（イミディエート）

SBC（キャリー付き減算（イミディエート））は、レジスタの値からイミディエート値とキャリーフラグの否定値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv6T2、ARMv7

SBC{S}<c> <Rd>,<Rn>,#<const>




```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

アセンブラ構文

```
SBC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<const>　　<Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

UAL 以前の構文の SBC<c>S は、SBCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

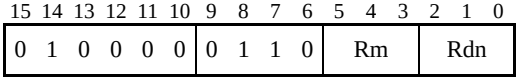
A6.7.108 SBC（レジスタ）

SBC（キャリー付き減算（レジスタ））は、レジスタの値から、オプションでシフトされたレジスタの値とキャリーフラグの否定値を減算して、結果をデスティネーションレジスタに書き込みます。オプションで、結果に基づいて条件フラグを更新することができます。

エンコード T1 RMv4T、ARMv5T*、ARMv6*、ARMv7

SBCS <Rdn>,<Rm> IT ブロック外部

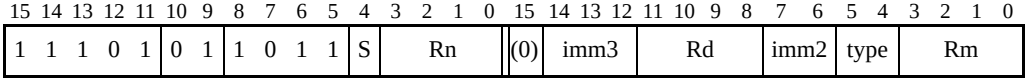
SBC<c> <Rdn>,<Rm> IT ブロック内部




```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

SBC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
SBC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　オプションでシフトされ、2 番目のオペランドとして使用されるレジスタを指定します。

<shift>　　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われず、両方のエンコードが許可されます。<shift> を指定した場合は、エンコード T2 のみが許可されます。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

UAL 以前の構文の SBC<c>S は、SBCS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

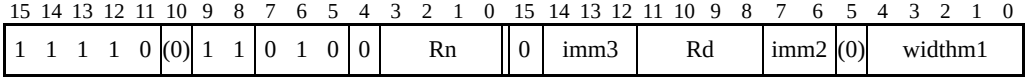
なし

A6.7.109 SBFX

この命令（符号付きビットフィールドの抽出）は、1 つのレジスタの任意の位置にある、任意の数の隣接するビットを抽出して 32 ビットに符号拡張し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

SBFX<c> <Rd>,<Rn>,#<lsb>,#<width>




```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

アセンブラ構文

```
SBFX<c><q>  <Rd>, <Rn>, #<lsb>, #<width>
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<Rn> 第一オペランドを格納しているレジスタを指定します。

<lsb> ビットフィールド内の最下位ビットのビット番号で、範囲は 0 〜 31 です。これによって、lsbit に必要な値が決定されます。

<width> ビットフィールドの幅で、範囲は 1 〜 32-<lsb> です。widthminus1 に必要な値は <width>-1 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = SignExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```

例外

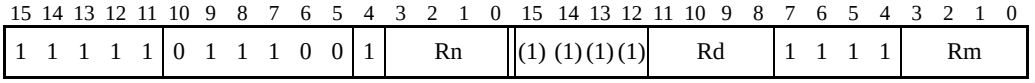
なし

A6.7.110 SDIV

SDIV（符号付き除算）は、32 ビットの符号付き整数レジスタの値を、32 ビットの符号付き整数レジスタの値で除算して、結果をデスティネーションレジスタに書き込みます。条件コードフラグは影響を受けません。

エンコード T1 プロファイル R および M のバージョンの Thumb-2 ISA

SDIV<c> <Rd>,<Rn>,<Rm>




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
SDIV<c><q>  {<Rd>,} <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

<Rn>　　被除数を含むレジスタを指定します。

<Rm>　　除数を含むレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if SInt(R[m]) == 0 then
        if IntegerZeroDivideTrappingEnabled() then
            GenerateIntegerZeroDivide();
        else
            result = 0;
    else
        result = RoundTowardsZero(SInt(R[n]) / SInt(R[m]));
    R[d] = result<31:0>;
```

例外

UsageFault

注

オーバフロー

符号付き整数除算の 0x80000000 / 0xFFFFFFFF が実行された場合は、擬似コードによって、32 ビット符号付き整数の範囲を超える、$+2^{31}$ の中間整数結果が生成されます。これによるオーバフローは発生しないため、R[d] に書き込む 32 ビットの結果を $+2^{31}$ のバイナリ表現の下位 32 ビットにする必要があります。そのため、除算の結果は 0x80000000 になります。

A6.7.111 SEV

SEV（イベント送信）はヒント命令です。マルチプロセッサシステム内部のすべての CPU に伝達されるイベントを発生させます。

この命令は NOP 互換ヒントです。詳細については、P.A6-16「*NOP 互換ヒント*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

SEV<c>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0

```
// No additional decoding required
```

エンコード T2 ARMv6T2、ARMv7

SEV<c>.W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	1	0	0

```
// No additional decoding required
```

アセンブラ構文

SEV<c><q>

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_SendEvent();
```

例外

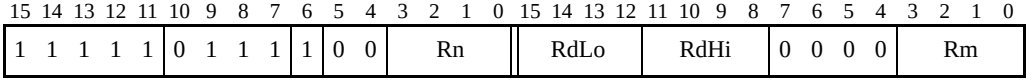
なし

A6.7.112 SMLAL

SMLAL（符号付きロング積和）は、2 つの符号付き 32 ビット値を乗算して 64 ビット値を生成し、別の 64 ビット値と加算します。

エンコード T1　　ARMv6T2、ARMv7

SMLAL<c> <RdLo>,<RdHi>,<Rn>,<Rm>




```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

アセンブラ構文

```
SMLAL<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<RdLo>　　加算される値の下位 32 ビットで、同時に結果の下位 32 ビットのデスティネーションレジスタとなります。

<RdHi>　　加算される値の上位 32 ビットで、同時に結果の上位 32 ビットのデスティネーションレジスタとなります。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　2 番目のオペランドを格納しているレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

例外

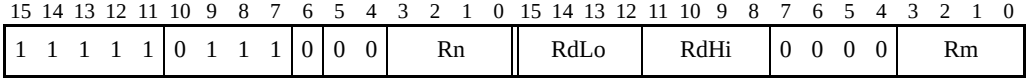
なし

A6.7.113 SMULL

SMULL（符号付きロング乗算）は、2 つの符号付き 32 ビット値を乗算して、64 ビットの結果を生成します。

エンコード T1　　ARMv6T2、ARMv7

SMULL<c> <RdLo>,<RdHi>,<Rn>,<Rm>




```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

アセンブラ構文

SMULL<c><q> <RdLo>, <RdHi>, <Rn>, <Rm>

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<RdLo>　　結果の下位 32 ビットが保存されます。

<RdHi>　　結果の上位 32 ビットが保存されます。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　2 番目のオペランドを格納しているレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

例外

なし

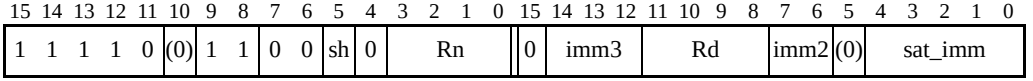
A6.7.114 SSAT

SSAT（符号付き飽和）は、オプションでシフトされる符号付きの値を、選択可能な符号付き範囲に飽和させます。

演算が飽和した場合は Q フラグがセットされます。

エンコード T1　　ARMv6T2、ARMv7

SSAT<c> <Rd>,#<imm5>,<Rn>{,<shift>}




```
if sh == '1' && (imm3:imm2) == '00000' then SEE SSAT16;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

アセンブラ構文

SSAT<c><q> <Rd>, #<imm>, <Rn> {,<shift>}

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<imm>　　飽和のビット位置を 1 ～ 32 の範囲で指定します。

<Rn>　　飽和させる値を格納しているレジスタを指定します。

<shift>　　オプションのシフトのビット数を指定します。<shift> を省略した場合、LSL #0 が使用されます。

指定する場合は、以下のいずれかにする必要があります。

LSL #N　N に許容される値の範囲は 0 ～ 31 です。

ASR #N　N に許容される値の範囲は 1 ～ 31 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, APSR.C);  // APSR.C ignored
    (result, sat) = SignedSatQ(SInt(operand), saturate_to);
    R[d] = SignExtend(result, 32);
    if sat then
        APSR.Q = '1';
```

例外

なし

A6.7.115 STC、STC2

STC、STC2（コプロセッサストア）は、コプロセッサから連続するメモリアドレスへデータをストアします。

コプロセッサが命令を実行できない場合は、UsageFault 例外が発生します。

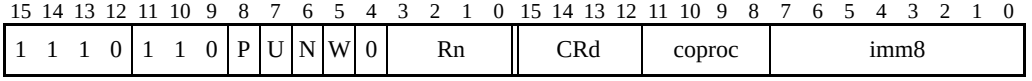
エンコード T1 すべての Thumb-2 ISA

STC{L}<c> <coproc>,<CRd>,[<Rn>{,#+/-<imm8>}]

STC{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm8>]!

STC{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm8>

STC{L}<c> <coproc>,<CRd>,[<Rn>],<option>




```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MCRR, MCRR2;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 then UNPREDICTABLE;
```

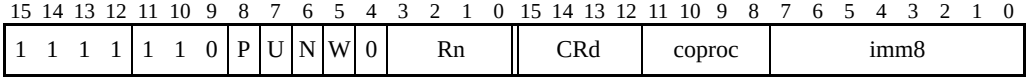
エンコード T2 すべての Thumb-2 ISA

STC2{L}<c> <coproc>,<CRd>,[<Rn>{,#+/-<imm8>}]

STC2{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm8>]!

STC2{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm8>

STC2{L}<c> <coproc>,<CRd>,[<Rn>],<option>




```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MCRR, MCRR2;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 then UNPREDICTABLE;
```

アセンブラ構文

STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>{,#+/-<imm>}]	オフセット：P = 1、W = 0
STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>,#+/-<imm>]!	プリインデクス：P = 1、W = 1
STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],#+/-<imm>	ポストインデクス：P = 0、W = 1
STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],<option>	インデクスなし：P = 0、W = 0、U = 1

各項目の説明については以下を参照して下さい。

2 指定した場合、エンコード T2 が選択されます。省略した場合、エンコード T1 が選択されます。

L 指定した場合、エンコードの N == 1 の形式が選択されます。省略した場合、N == 0 の形式が選択されます。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<coproc> コプロセッサの名前を指定します。標準の汎用コプロセッサ名は、p0、p1、...、p15 です。

<CRd> コプロセッサのソースレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために <Rn> の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は、4 の倍数で 0 〜 1020 です。オフセットアドレシング構文では <imm> を省略可能で、その場合オフセットは 0 と見なされます。

<option> コプロセッサに対する追加のオプションを、{ と } で囲んだ 0 〜 255 の範囲の整数として指定します。この整数は、命令の imm8 フィールドにエンコードされます。

UAL 以前の構文の STC<c>L は、STCL<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        repeat
            MemA[address,4] = Coproc_GetWordToStore(cp, ThisInstr());  address = address + 4;
        until Coproc_DoneStoring(cp, ThisInstr());
        if wback then R[n] = offset_addr;
```

例外

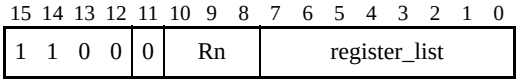
UsageFault、MemManage、BusFault

A6.7.116 STM、STMIA、STMEA

STM、STMIA、STMEA（ポストインクリメント複数ストア（空上昇複数ストア））は、ベースレジスタからのアドレスを使用して、複数のレジスタを連続するメモリ位置にストアします。連続するメモリ位置は、このアドレスから開始されます。オプションとして、ストアされる最後のメモリ位置の次のアドレスをベースアドレスに書き戻すことができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

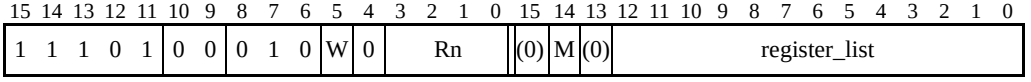
STM<c> <Rn>!,<registers>




```
n = UInt(Rn);  registers = '00000000':register_list;  wback = TRUE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

エンコード T2 ARMv6T2、ARMv7

STM<c>.W <Rn>{!},<registers>




```
n = UInt(Rn);  registers = '0':M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

アセンブラ構文

```
STM{IA|EA}<c><q>  <Rn>{!}, <registers>
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　　ベースレジスタを指定します。SP を指定できます。

!　　命令によって、変更された値が <Rn> に書き戻されます。! を省略した場合、命令によって <Rn> は変更されません。

<registers>

1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストで、ストアされるレジスタのセットを指定します。最も小さい番号のレジスタは最下位のメモリアドレスにストアされ、以下順にストアが行われ、最も大きい番号のレジスタは最上位のメモリアドレスにストアされます。

エンコード T2 では、1 つのレジスタしか含まないリストはサポートされません。リスト内に <Rt> のみを含む STMIA 命令が Thumb にアセンブルされ、エンコード T1 が使用できない場合は、その命令と等価な STR<c><q> <Rt>,[<Rn>]{,#-4} 命令にアセンブルされます。

リストに SP と PC を含めることはできません。

STMEA と STMIA は STM の同義語で、STMEA は空上昇スタックにデータをプッシュするために使用されることを示しています。

UAL 以前の構文の STMSTM<c>IA および STM<c>EA は、STM<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];

    for i = 0 to 14
        if i == n && wback && i != LowestSetBit(registers) then
            MemA[address,4] = bits(32) UNKNOWN;                // encoding T1 only
        else
            MemA[address,4] = R[i];
        address = address + 4;

    if wback then R[n] = R[n] + 4*BitCount(registers);
```

例外

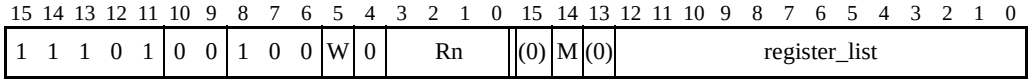
UsageFault、MemManage、BusFault

A6.7.117 STMDB、STMFD

STMDB、STMFD（プリデクリメント複数ストア（フル下降複数ストア））は、ベースレジスタからのアドレスを使用して、複数のレジスタを連続するメモリ位置にストアします。連続するメモリ位置は、このアドレスの 1 つ前で終了します。オプションとして、ロードされる最初のメモリ位置のアドレスをベースアドレスに書き戻すことができます。

エンコード T1 ARMv6T2、ARMv7

STMDB<c> <Rn>{!},<registers>




```
if W == '1' && Rn == '1101' then SEE PUSH;
n = UInt(Rn);  registers = '0':M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

アセンブラ構文

STMDB<c><q> <Rn>{!}, <registers>

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn> ベースレジスタを指定します。SP を指定できます。SP を指定し、! を指定した場合は、P.A6-187「*PUSH*」に説明されているように扱われます。

! W ビットが 1 にセットされ、変更された値が <Rn> に書き戻されます。! を省略した場合、命令によって <Rn> は変更されません。

<registers>

1 つ以上のレジスタがカンマで区切られ、{ と } で囲まれたリストで、ストアされるレジスタのセットを指定します。番号が最も小さいレジスタが最下位のメモリアドレスにストアされ、以下順にストアが行われ、番号が最も大きいレジスタが最上位のメモリアドレスにストアされます。

エンコード T1 では、1 つのレジスタのみを含むリストはサポートされません。リストに <Rt> のみを含む STMDB 命令が Thumb にアセンブルされた場合、その命令と等価な STR<c><q> <Rt>,[<Rn>,#-4]{!} 命令にアセンブルされます。

リストに SP と PC を含めることはできません。

STMFD は STMDB の同義語で、フル下降スタックにデータをプッシュするために使用されることを示しています。

UAL 以前の構文の STM<c>DB および STM<c>FD は、STMDB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);

    for i = 0 to 14
        if registers<i> == '1' then
            MemA[address,4] = R[i];
            address = address + 4;

    if wback then R[n] = R[n] - 4*BitCount(registers);
```

例外

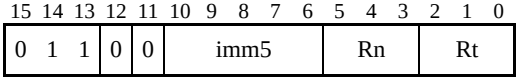
UsageFault、MemManage、BusFault

A6.7.118 STR（イミディエート）

STR（レジスタストア（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 ワードをストアします。オフセットアドレシング、ポストインデクスアドレシング、プリインデクスアドレシングを使用できます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

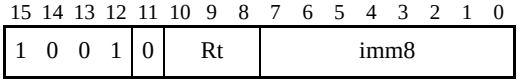
STR<c> <Rt>, [<Rn>{,#<imm5>}]




```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

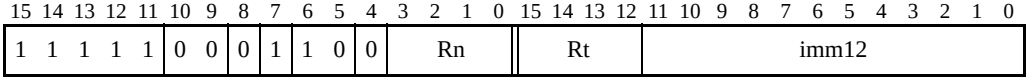
STR<c> <Rt>,[SP,#<imm8>]




```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T3 ARMv6T2、ARMv7

STR<c>.W <Rt>,[<Rn>,#<imm12>]



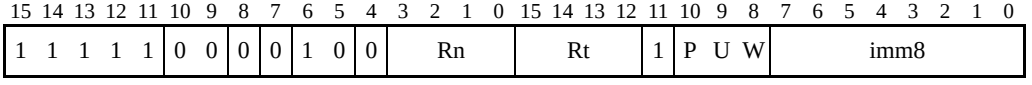

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE;
```

エンコード T4 ARMv6T2、ARMv7

STR<c> <Rt>,[<Rn>,#-<imm8>]

STR<c> <Rt>,[<Rn>],#+/-<imm8>

STR<c> <Rt>,[<Rn>,#+/-<imm8>]!




```
if P == '1' && U == '1' && W == '0' then SEE STRT;
if Rn == '1101' && P == '1' && U == '0' && W == '1' && imm8 == '00000100' then SEE PUSH;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```


アセンブラ構文

STR<c><q> <Rt>, [<Rn> {, #+/-<imm>}]	オフセット：index==TRUE、wback==FALSE
STR<c><q> <Rt>, [<Rn>, #+/-<imm>]!	プリインデクス：index==TRUE、wback==TRUE
STR<c><q> <Rt>, [<Rn>], #+/-<imm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> ソースレジスタを指定します。SP を指定できます。

<Rn> ベースレジスタを指定します。SP を指定できます。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために <Rn> の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は、エンコード T1 では 4 の倍数で 0 ～ 124、エンコード T2 では 4 の倍数で 0 ～ 1020、エンコード T3 では 0 ～ 4095 の任意の値、エンコード T4 では 0 ～ 255 の任意の値です。オフセットアドレシング構文では <imm> を省略可能で、その場合オフセットは 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,4] = R[t];
    if wback then R[n] = offset_addr;
```

例外

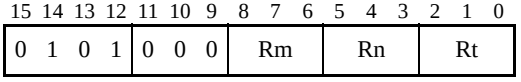
UsageFault、MemManage、BusFault

A6.7.119 STR（レジスタ）

STR（レジスタストア（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、レジスタからそのメモリに 1 ワードをストアします。オフセットレジスタの値は、左に 0、1、2、3 ビットシフトすることができます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

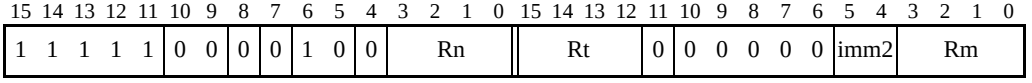
STR<c> <Rt>,[<Rn>,<Rm>]




```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

STR<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]




```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

STR<c><q> <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> ソースレジスタを指定します。SP を指定できます。

<Rn> ベース値を含むレジスタを指定します。SP を指定できます。

<Rm> アドレス生成のために左にシフトして <Rn> の値に加算されるオフセットが含まれます。

<shift> <Rm> からの値を左にシフトするビット数を指定します。範囲は 0 ～ 3 です。このオプションを省略した場合、0 ビットのシフトと見なされ、両方のエンコードが許可されます。このオプションを指定した場合は、エンコード T2 のみが許可されます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    data = R[t];
    MemU[address,4] = data;
```

例外

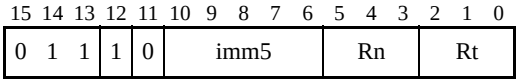
UsageFault、MemManage、BusFault

A6.7.120 STRB（イミディエート）

STRB（レジスタストア バイト（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 バイトをストアします。オフセットアドレシング、ポストインデクスアドレシング、プリインデクスアドレシングを使用できます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

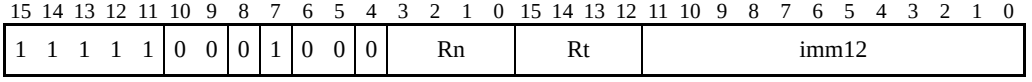
STRB<c> <Rt>,[<Rn>,#<imm5>]




```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T2 ARMv6T2、ARMv7

STRB<c>.W <Rt>,[<Rn>,#<imm12>]



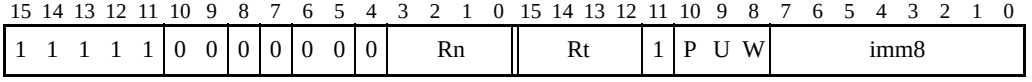

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if BadReg(t) then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

STRB<c> <Rt>,[<Rn>,#-<imm8>]

STRB<c> <Rt>,[<Rn>],#+/-<imm8>

STRB<c> <Rt>,[<Rn>,#+/-<imm8>]!




```
if P == '1' && U == '1' && W == '0' then SEE STRBT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```


アセンブラ構文

STRB<c><q> <Rt>, [<Rn> {, #+/-<imm>}]	オフセット：index==TRUE、wback==FALSE
STRB<c><q> <Rt>, [<Rn>, #+/-<imm>]!	プリインデクス：index==TRUE、wback==TRUE
STRB<c><q> <Rt>, [<Rn>], #+/-<imm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> ソースレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために <Rn> の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は、T1 エンコードでは 0 ～ 31、T2 エンコードでは 0 ～ 4095、T3 エンコードでは 0 ～ 255 です。オフセットアドレシング構文では <imm> を省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の STR<c>B は、STRB<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

例外

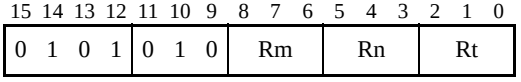
MemManage、BusFault

A6.7.121 STRB（レジスタ）

STRB（レジスタストア バイト（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、レジスタからそのメモリに 1 バイトをストアします。オフセットレジスタの値は、左に 0、1、2、3 ビットシフトできます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

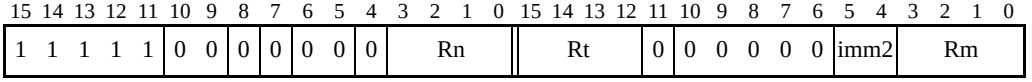
STRB<c> <Rt>,[<Rn>,<Rm>]




```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

STRB<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]




```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(t) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
STRB<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　ベース値を含むレジスタを指定します。SP を指定できます。

<Rm>　アドレス生成のために左にシフトされ、<Rn> の値に加算されるオフセットが含まれます。

<shift>　<Rm> からの値を左にシフトするビット数を指定します。範囲は 0 ～ 3 です。このオプションを省略した場合、0 ビットのシフトと見なされ、両方のエンコードが許可されます。このオプションを指定した場合は、エンコード T2 のみが許可されます。

UAL 以前の構文の STR<c>B は、STRB<c> と等価です

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,1] = R[t]<7:0>;
```

例外

MemManage、BusFault

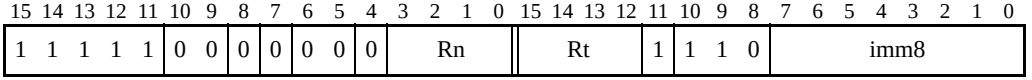
A6.7.122 STRBT

STRBT（非特権レジスタストア バイト）は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 バイトをストアします。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

プロセッサが非特権状態で実行している場合と同様にメモリアクセスが制限されます（実際にプロセッサが非特権状態で実行している場合と違いはありません）。

エンコード T1 ARMv6T2、ARMv7

STRBT<c> <Rt>,[<Rn>,#<imm8>]




```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```


アセンブラ構文

```
STRBT<c><q>  <Rt>, [<Rn> {, #<imm>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt>　　ソースレジスタを指定します。

<Rn>　　ベースレジスタを指定します。SP を指定できます。

<imm>　　アドレス生成のために <Rn> の値に加算されるイミディエートオフセットを指定します。許容される値の範囲は 0 ～ 255 です。<imm> は省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の STR<c>BT は、STRBT<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    MemU_unpriv[address,1] = R[t]<7:0>;
```

例外

MemManage、BusFault

A6.7.123 STRD（イミディエート）

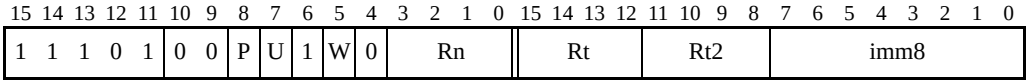
STRD（デュアルレジスタ ストア（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、2 つのレジスタからそのメモリに 2 ワードをストアします。オフセットアドレシング、ポストインデクスアドレシング、プリインデクスアドレシングを使用できます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

STRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm8>}]

STRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm8>

STRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm8>]!




```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if n == 15 || BadReg(t) || BadReg(t2) then UNPREDICTABLE;
```

関連エンコード P.A5-21「*ダブルワードのロード/ ストア、排他ロード/ ストア、テーブル分岐*」を参照

アセンブラ構文

STRD<c><q> <Rt>,<Rt2>,[<Rn>{,#+/-<imm>}]	オフセット：index==TRUE、wback==FALSE
STRD<c><q> <Rt>,<Rt2>,[<Rn>,#+/-<imm>]!	プリインデクス：index==TRUE、wback==TRUE
STRD<c><q> <Rt>,<Rt2>,[<Rn>],#+/-<imm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> 最初のソースレジスタを指定します。

<Rt2> 2 番目のソースレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために <Rn> の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は、4 の倍数で 0 ～ 1020 です。オフセットアドレシング構文では <imm> を省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の STR<c>D は、STRD<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemA[address,4] = R[t];
    MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```

例外

UsageFault、MemManage、BusFault

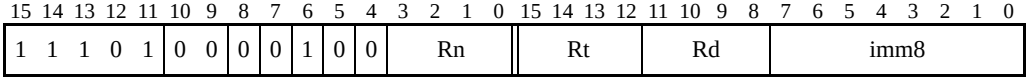
A6.7.124 STREX

STREX（排他レジスタストア）は、実行中のプロセッサがアドレス指定されたメモリに対して排他アクセス権を持っている場合に、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 ワードをストアします。

メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

STREX<c> <Rd>,<Rt>,[<Rn>{,#<imm8>}]




```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if BadReg(d) || BadReg(t) || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

アセンブラ構文

STREX<c><q> <Rd>, <Rt>, [<Rn> {,#<imm>}]

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> ステータス値が返されるデスティネーションレジスタ。返される値は次のとおりです。

0 命令によってメモリが更新された場合

1 命令によってメモリが更新されなかった場合

<Rt> ソースレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。

<imm> アドレス生成のために <Rn> の値に加算されるイミディエートオフセットを指定します。許容される値の範囲は、4 の倍数で 0 ～ 1020 です。<imm> は省略可能で、その場合オフセットは 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    if ExclusiveMonitorsPass(address,4) then
        MemA[address,4] = R[t];
        R[d] = 0;
    else
        R[d] = 1;
```

例外

UsageFault、MemManage、BusFault

A6.7.125 STREXB

STREXB（排他レジスタストア バイト）は、実行中のプロセッサがアドレス指定されたメモリに対して排他アクセス権を持っている場合に、ベースレジスタの値からアドレスを導出して、レジスタからそのメモリへ 1 バイトをストアします。

メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6K またはそれ以降のすべてのバージョンの Thumb ISA

STREXB<c> <Rd>,<Rt>,[<Rn>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn				Rt				(1)	(1)	(1)	(1)	0	1	0	0	Rd			

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if BadReg(d) || BadReg(t) || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

アセンブラ構文

STREXB<c><q> <Rd>, <Rt>, [<Rn>]

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> ステータス値が返されるデスティネーションレジスタ。返される値は次のとおりです。
- **0** 命令によってメモリが更新された場合
- **1** 命令によってメモリが更新されなかった場合

<Rt> ソースレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if ExclusiveMonitorsPass(address,1) then
        MemA[address,1] = R[t];
        R[d] = 0;
    else
        R[d] = 1;
```

例外

MemManage、BusFault

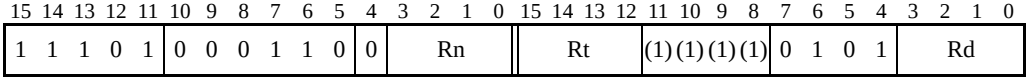
A6.7.126 STREXH

STREXH（排他レジスタストア ハーフワード）は、実行中のプロセッサがアドレス指定されたメモリに対して排他アクセス権を持っている場合に、ベースレジスタの値からアドレスを導出して、レジスタからそのメモリへ 1 ハーフワードをストアします。

メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv6K またはそれ以降のすべてのバージョンの Thumb ISA

STREXH<c> <Rd>,<Rt>,[<Rn>]




```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if BadReg(d) || BadReg(t) || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```


アセンブラ構文

```
STREXH<c><q>  <Rd>, <Rt>, [<Rn>]
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　ステータス値が返されるデスティネーションレジスタ。返される値は次のとおりです。
0　　命令によってメモリが更新された場合
1　　命令によってメモリが更新されなかった場合

<Rt>　　ソースレジスタを指定します。

<Rn>　　ベースレジスタを指定します。SP を指定できます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if ExclusiveMonitorsPass(address,2) then
        MemA[address,2] = R[t];
        R[d] = 0;
    else
        R[d] = 1;
```

例外

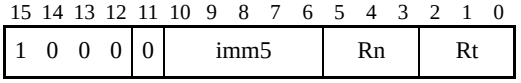
UsageFault、MemManage、BusFault

A6.7.127 STRH（イミディエート）

STRH（レジスタストア ハーフワード（イミディエート））は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 ハーフワードをストアします。オフセットアドレシング、ポストインデクスアドレシング、プリインデクスアドレシングを使用できます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

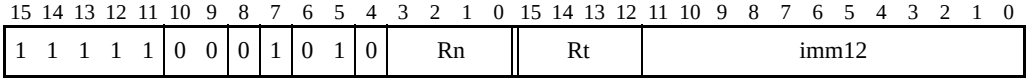
STRH<c> <Rt>,[<Rn>{,#<imm5>}]




```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

エンコード T2 ARMv6T2、ARMv7

STRH<c>.W <Rt>,[<Rn>{,#<imm12>}]



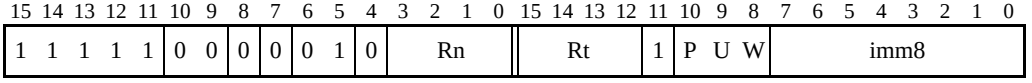

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if BadReg(t) then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

STRH<c> <Rt>,[<Rn>,#-<imm8>]

STRH<c> <Rt>,[<Rn>],#+/-<imm8>

STRH<c> <Rt>,[<Rn>,#+/-<imm8>]!




```
if P == '1' && U == '1' && W == '0' then SEE STRHT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if BadReg(t) || (wback && n == t) then UNPREDICTABLE;
```


アセンブラ構文

STRH<c><q> <Rt>, [<Rn> {, #+/-<imm>}]	オフセット：index==TRUE、wback==FALSE
STRH<c><q> <Rt>, [<Rn>, #+/-<imm>]!	プリインデクス：index==TRUE、wback==TRUE
STRH<c><q> <Rt>, [<Rn>], #+/-<imm>	ポストインデクス：index==FALSE、wback==TRUE

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> ソースレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。

+/- イミディエートオフセットをベースレジスタの値に加算する場合 (add == TRUE) は、+ を指定するか、省略します。減算する場合 (add == FALSE) は、- を指定します。#0 と #-0 では異なる命令が生成されます。

<imm> アドレス生成のために <Rn> の値に対して加算または減算されるイミディエートオフセットを指定します。許容される値の範囲は、エンコード T1 では 2 の倍数で 0 ～ 62、エンコード T2 では 0 ～ 4095 の任意の値、エンコード T3 では 0 ～ 255 の任意の値です。オフセットアドレシング構文では <imm> を省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の STR<c>H は、STRH<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,2] = R[t]<15:0>;

    if wback then R[n] = offset_addr;
```

例外

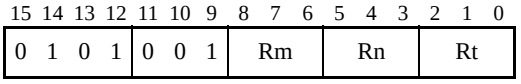
UsageFault、MemManage、BusFault

A6.7.128 STRH（レジスタ）

STRH（レジスタストア ハーフワード（レジスタ））は、ベースレジスタの値とオフセットレジスタの値からアドレスを計算して、レジスタからそのメモリに 1 ハーフワードをストアします。オフセットレジスタの値は、左に 0、1、2、3 ビットシフトできます。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

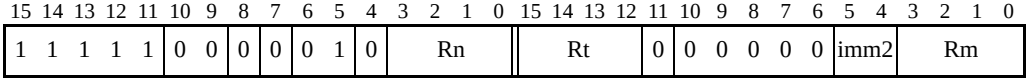
STRH<c> <Rt>,[<Rn>,<Rm>]




```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

STRH<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]




```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if BadReg(t) || BadReg(m) then UNPREDICTABLE;
```


アセンブラ構文

```
STRH<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　ベース値を含むレジスタを指定します。SP を指定できます。

<Rm>　アドレス生成のために左にシフトされて、<Rn> の値に加算されるオフセットが含まれます。

<shift>　<Rm> からの値を左にシフトするビット数を指定します。範囲は 0 ～ 3 です。このオプションを省略した場合は、0 ビットのシフトと見なされ、両方のエンコードが許可されます。このオプションを指定した場合は、エンコード T2 のみが許可されます。

UAL 以前の構文の STR<c>H は、STRH<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,2] = R[t]<15:0>;
```

例外

UsageFault、MemManage、BusFault

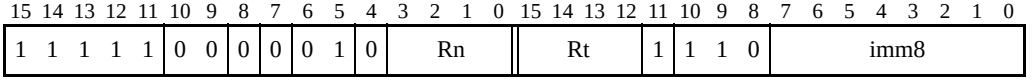
A6.7.129 STRHT

STRHT（非特権レジスタストア ハーフワード）は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 ハーフワードをストアします。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

プロセッサが非特権状態で実行している場合と同様にメモリアクセスが制限されます（実際にプロセッサが非特権状態で実行している場合と違いはありません）。

エンコード T1 ARMv6T2、ARMv7

STRHT<c> <Rt>,[<Rn>,#<imm8>]




```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

アセンブラ構文

STRHT<c><q> <Rt>, [<Rn> {, #<imm>}]

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> ソースレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。

<imm> アドレス生成のために <Rn> の値に加算されるイミディエートオフセットを指定します。許容される値の範囲は 0 ～ 255 です。<imm> は省略可能で、その場合オフセットは 0 と見なされます。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    MemU_unpriv[address,2] = R[t]<15:0>;
```

例外

UsageFault、MemManage、BusFault

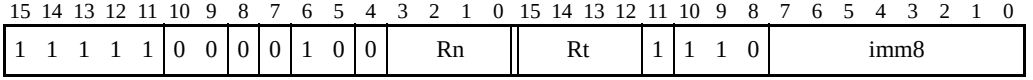
A6.7.130 STRT

STRT（非特権レジスタストア）は、ベースレジスタの値とイミディエートオフセットからアドレスを計算して、レジスタからそのメモリに 1 ワードをストアします。メモリアクセスの詳細については、P.A6-15「*メモリアクセス*」を参照して下さい。

プロセッサが非特権状態で実行している場合と同様にメモリアクセスが制限されます(実際にプロセッサが非特権状態で実行している場合と違いはありません)。

エンコード T1 ARMv6T2、ARMv7

STRT<c> <Rt>,[<Rn>,#<imm8>]




```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if BadReg(t) then UNPREDICTABLE;
```

アセンブラ構文

```
STRT<c><q>  <Rt>, [<Rn> {, #<imm>}]
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rt> ソースレジスタを指定します。

<Rn> ベースレジスタを指定します。SP を指定できます。

<imm> アドレス生成のために <Rn> の値に加算されるイミディエートオフセットを指定します。許容される値の範囲は 0 ～ 255 です。<imm> は省略可能で、その場合オフセットは 0 と見なされます。

UAL 以前の構文の STR<c>T は、STRT<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    data = R[t];
    MemU_unpriv[address,4] = data;
```

例外

UsageFault、MemManage、BusFault

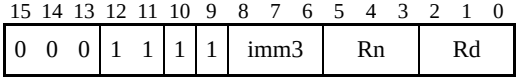
A6.7.131 SUB（イミディエート）

SUB（減算（イミディエート））は、レジスタの値からイミディエート値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

SUBS <Rd>,<Rn>,#<imm3> IT ブロック外部

SUB<c> <Rd>,<Rn>,#<imm3> IT ブロック内部



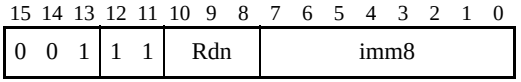
15 14 13	12 11	10	9	8 7 6	5 4 3	2 1 0
0 0 0	1 1	1	1	imm3	Rn	Rd

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

SUBS <Rdn>,#<imm8> IT ブロック外部

SUB<c> <Rdn>,#<imm8> IT ブロック内部

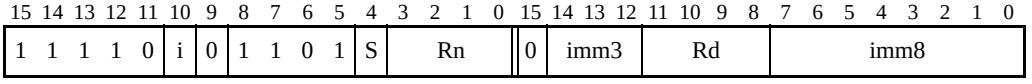


15 14 13	12 11	10 9 8	7 6 5 4 3 2 1 0
0 0 1	1 1	Rdn	imm8

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

エンコード T3 ARMv6T2、ARMv7

SUB{S}<c>.W <Rd>,<Rn>,#<const>

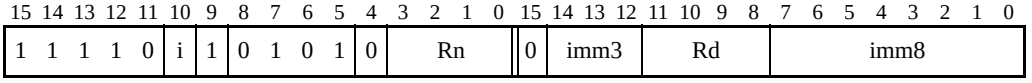


15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12 11	10 9 8 7	6 5 4 3 2 1 0
1 1 1 1 0	i	0	1 1 0 1	S	Rn	0	imm3	Rd	imm8

```
if Rd == '1111' && setflags then SEE CMP (immediate);
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if BadReg(d) || n == 15 then UNPREDICTABLE;
```

エンコード T4 ARMv6T2、ARMv7

SUBW<c> <Rd>,<Rn>,#<imm12>



15 14 13 12 11	10	9	8 7 6 5	4	3 2 1 0	15	14 13 12 11	10 9 8 7	6 5 4 3 2 1 0
1 1 1 1 0	i	1	0 1 0 1	0	Rn	0	imm3	Rd	imm8

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if BadReg(d) then UNPREDICTABLE;
```

アセンブラ構文

SUB{S}<c><q> {<Rd>,} <Rn>, #<const>	すべてのエンコードが許可される
SUBW<c><q> {<Rd>,} <Rn>, #<const>	エンコード T4 のみが許可される

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

<Rn>　第一オペランドを格納しているレジスタを指定します。<Rn> に SP を指定した場合については、P.A6-247「*SUB (SP - イミディエート)*」を参照して下さい。<Rn> に PC を指定した場合については、P.A6-30「*ADR*」を参照して下さい。

<const>　<Rn> から取得された値から減算されるイミディエート値を指定します。許容される値の範囲は、エンコード T1 では 0 ～ 7、エンコード T2 では 0 ～ 255、エンコード T4 では 0 ～ 4095 です。エンコード T3 で許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

同じ長さの複数のエンコードが使用可能な場合は、エンコード T4 よりもエンコード T3 の方が適しています（エンコード T4 が必要な場合は、SUBW 構文を使用して下さい）。<Rd> を指定する場合は、エンコード T2 よりもエンコード T1 の方が適しています。<Rd> を省略する場合は、エンコード T1 よりもエンコード T2 の方が適しています。

UAL 以前の構文の SUB<c>S は、SUBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

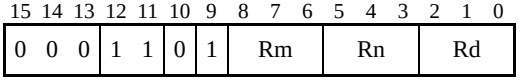
A6.7.132 SUB（レジスタ）

SUB（減算（レジスタ））は、レジスタの値から、オプションでシフトされるレジスタの値を減算して、結果をデスティネーションレジスタに書き込みます。オプションとして、結果に基づいて条件フラグを更新することができます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

SUBS <Rd>,<Rn>,<Rm> IT ブロック外部

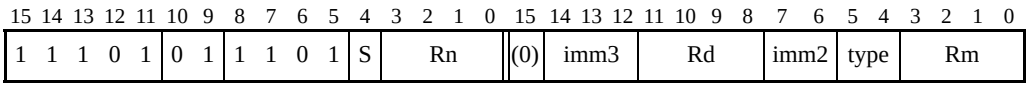
SUB<c> <Rd>,<Rn>,<Rm> IT ブロック内部




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv4T、ARMv5T*、ARMv6*、ARMv7

SUB{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}




```
if Rd == '1111' && S == '1' then SEE CMP (register);
if Rn == '1101' then SEE SUB (SP minus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(d) || n == 15 || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
SUB{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

<Rn>　　第一オペランドを格納しているレジスタを指定します。<Rn> に SP を指定した場合については、P.A6-249「*SUB (SP - レジスタ)*」を参照して下さい。

<Rm>　　オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタを指定します。

<shift>　　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われず、両方のエンコードが許可されます。<shift> を指定した場合は、エンコード T2 のみが許可されます。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

UAL 以前の構文の SUB<c>S は、SUBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

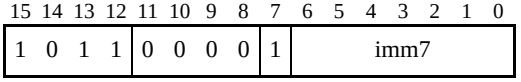
なし

A6.7.133 SUB（SP - イミディエート）

SUB（減算（SP - イミディエート））は、SP の値からイミディエート値を減算して、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

SUB<c> SP,SP,#<imm7>

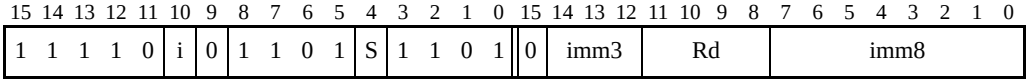



```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

エンコード T2 ARMv6T2、ARMv7

SUB{S}<c>.W <Rd>,SP,#<const>

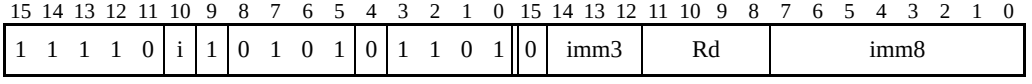
T2 SUB{S}<c>.W <Rd>,SP,#<const>




```
if Rd == '1111' && S == '1' then SEE CMP (immediate);
d = UInt(Rd);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 15 then UNPREDICTABLE;
```

エンコード T3 ARMv6T2、ARMv7

SUBW<c> <Rd>,SP,#<imm12>




```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

アセンブラ構文

SUB{S}<c><q> {<Rd>,} SP, #<const>	すべてのエンコードが許可される
SUBW<c><q> {<Rd>,} SP, #<const>	エンコード T4 のみが許可される

各項目の説明については以下を参照して下さい。

S　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。<Rd> を省略した場合、SP が使用されます。

<const>　<Rn> から取得された値に加算されるイミディエート値を指定します。オフセットに許容される値の範囲は、エンコード T1 では 4 の倍数で 0 ～ 508、エンコード T3 では 0 ～ 4095 の任意の値です。エンコード T2 で許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

両方の 32 ビットエンコードが使用可能な場合は、エンコード T3 よりもエンコード T2 の方が適しています（エンコード T3 が必要な場合は、SUBW 構文を使用して下さい）。

UAL 以前の構文の SUB<c>S は、SUBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, NOT(imm32), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

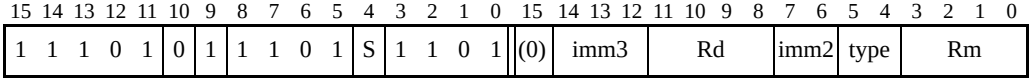
なし

A6.7.134 SUB（SP - レジスタ）

SUB（減算（SP - レジスタ））は、SP の値から、オプションでシフトされるレジスタの値を減算して、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

SUB{S}<c> <Rd>,SP,<Rm>{,<shift>}




```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 && (shift_t != SRType_LSL || shift_n > 3) then UNPREDICTABLE;
if d == 15 || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
SUB{S}<c><q>  {<Rd>,} SP, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

S　　指定した場合、命令はフラグを更新します。それ以外の場合、命令はフラグを更新しません。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合、SP が使用されます。

<Rm>　　オプションでシフトされる、2 番目のオペランドとして使用されるレジスタを指定します。

<shift>　　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われません。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

<Rd> に SP を指定した、または省略した場合、<shift> に使用できるのは LSL #0、LSL #1、LSL #2、LSL #3 のみです。

UAL 以前の構文の SUB<c>S は、SUBS<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(SP, NOT(shifted), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

例外

なし

A6.7.135 SVC（以前の SWI）

SWI は、スーパバイザコールを発生させます。詳細については、*『ARM アーキテクチャ リファレンスマニュアル』*の*「例外」*を参照して下さい。

サービスを提供するオペレーティングシステムへの呼び出しとして使用します。

エンコード T1　　すべてのバージョンの Thumb ISA

SVC<c> #<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	imm8							

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly, and is ignored by hardware. SVC handlers in some
// systems interpret imm8 in software, for example to determine the required service.
```

アセンブラ構文

```
SVC<c><q>  #<imm>
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<imm>　　8 ビットのイミディエート定数を指定します。

UAL 以前の構文の SWI<c> は、SVC<c> と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CallSupervisor();
```

例外

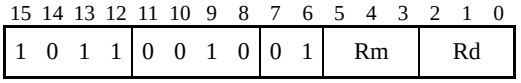
SVCall

A6.7.136 SXTB

SXTB（符号付きバイト拡張）は、レジスタから 8 ビット値を抽出して、32 ビットに符号拡張し、結果をデスティネーションレジスタに書き込みます。8 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6 またはそれ以降のすべてのバージョンの Thumb ISA

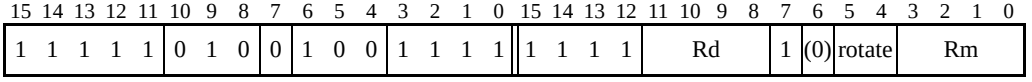
SXTB<c> <Rd>,<Rm>




```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

エンコード T2 ARMv6T2、ARMv7

SXTB<c>.W <Rd>,<Rm>{,<rotation>}




```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
SXTB<c><q>  <Rd>, <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rm>　　オペランドを含むレジスタを指定します。

<rotation>

次のいずれかのローテーションを指定できます。

- ROR #8.
- ROR #16.
- ROR #24.
- 省略

注

お使いのアセンブラで #0 によるシフトが記述でき、それがシフトなしまたは LSL #0 と等価として処理される場合、このオプションの ROR #0 も記述できるはずです。これは、<rotation> を省略した場合と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<7:0>, 32);
```

例外

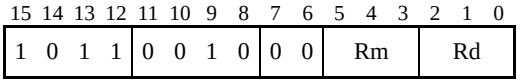
なし

A6.7.137 SXTH

この命令（符号付きハーフワード拡張）は、レジスタから 16 ビット値を抽出して、32 ビットに符号拡張し、結果をデスティネーションレジスタに書き込みます。16 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6 またはそれ以降のすべてのバージョンの Thumb ISA

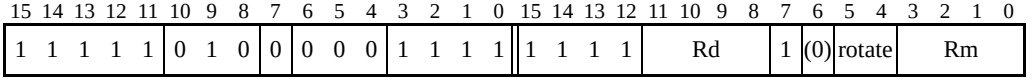
SXTH<c> <Rd>,<Rm>




```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

エンコード T2 ARMv6T2、ARMv7

SXTH<c>.W <Rd>,<Rm>{,<rotation>}




```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
SXTH<c><q>  <Rd>, <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。

<Rm>　　オペランドを含むレジスタを指定します。

<rotation>

次のいずれかのローテーションを指定できます。

- ROR #8.
- ROR #16.
- ROR #24.
- 省略

注

お使いのアセンブラで #0 によるシフトが記述でき、それがシフトなしまたは LSL #0 と等価として処理される場合、このオプションの ROR #0 も記述できるはずです。これは、<rotation> を省略した場合と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);
```

例外

なし

A6.7.138 TBB、TBH

TBB（テーブル分岐バイト）は、単一バイトオフセットのテーブルを使用した、PC 相対の前方分岐を発生させます。ベースレジスタはテーブルへのポインタとして、2 番目のレジスタはテーブル内のインデクスとして使用されます。分岐の長さは、テーブルから返されたバイト値の 2 倍です。

TBH（テーブル分岐ハーフワード）は、単一ハーフワードオフセットのテーブルを使用した、PC 相対の前方分岐を発生させます。ベースレジスタはテーブルへのポインタとして、2 番目のレジスタはテーブル内のインデクスとして使用されます。分岐の長さは、テーブルから返されたハーフワード値の 2 倍です。

エンコード T1 ARMv6T2、ARMv7

TBB<c> [<Rn>,<Rm>]
IT ブロック外部または IT ブロック内部の最後

TBH<c> [<Rn>,<Rm>,LSL #1]
IT ブロック外部または IT ブロック内部の最後

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	H	Rm			

```
n = UInt(Rn);  m = UInt(Rm);  is_tbh = (H == '1');
if n == 13 || BadReg(m) then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

アセンブラ構文

```
TBB<c><q>  [<Rn>, <Rm>]

TBH<c><q>  [<Rn>, <Rm>, LSL #1]
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　　ベースレジスタ。このレジスタには、分岐長のテーブルのアドレスが含まれます。PC を指定できます。その場合は、この命令の直後にテーブルが続きます。

<Rm>　　インデクスレジスタ。

TBB の場合は、テーブル内の 1 バイトを指す整数が含まれます。テーブル内のオフセットはインデクスの値です。

TBH の場合は、テーブル内の 1 ハーフワードを指す整数が含まれます。テーブル内のオフセットはインデクスの値の 2 倍です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if is_tbh then
        halfwords = UInt(MemU[R[n]+LSL(R[m],1), 2]);
    else
        halfwords = UInt(MemU[R[n]+R[m], 1]);
    BranchWritePC(PC + 2*halfwords);
```

例外

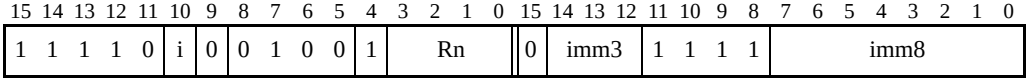
MemManage、BusFault

A6.7.139 TEQ（イミディエート）

TEQ（等価テスト（イミディエート））は、レジスタの値とイミディエート値に対して排他的論理和演算を実行します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv6T2、ARMv7

TEQ<c> <Rn>,#<const>




```
n = UInt(Rn);
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(n) then UNPREDICTABLE;
```

アセンブラ構文

```
TEQ<c><q>  <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn> オペランドを含むレジスタを指定します。

<const> <Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

例外

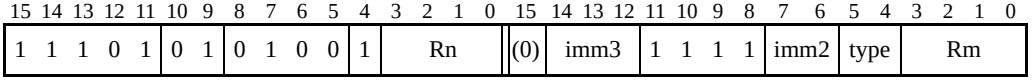
なし

A6.7.140 TEQ（レジスタ）

TEQ（等価テスト（レジスタ））は、レジスタの値と、オプションとしてシフトされたレジスタの値との間で排他的論理和演算を実行します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1　　ARMv6T2、ARMv7

```
TEQ<c> <Rn>,<Rm>{,<shift>}
```




```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
TEQ<c><q>  <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　オプションでシフトされる、2 番目のオペランドとして使用されるレジスタを指定します。

<shift>　　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われません。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

例外

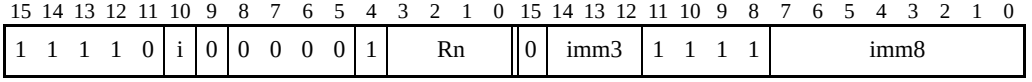
なし

A6.7.141 TST（イミディエート）

TST（テスト（イミディエート））は、レジスタの値とイミディエート値との間で論理積演算を実行します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv6T2、ARMv7

TST<c> <Rn>,#<const>




```
n = UInt(Rn);
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if BadReg(n) then UNPREDICTABLE;
```

アセンブラ構文

```
TST<c><q>  <Rn>, #<const>
```

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　　オペランドを含むレジスタを指定します。

<const>　　<Rn> から取得された値に加算されるイミディエート値を指定します。許容される値の範囲については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

例外

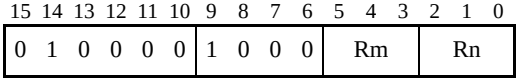
なし

A6.7.142 TST（レジスタ）

TST（テスト（レジスタ））は、レジスタの値と、オプションでシフトされるレジスタの値との間で論理積演算を実行します。結果に基づいて条件フラグを更新し、結果を破棄します。

エンコード T1 ARMv4T、ARMv5T*、ARMv6*、ARMv7

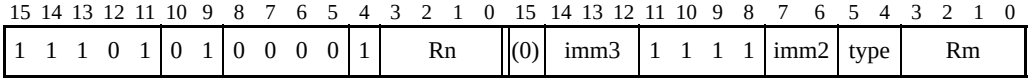
TST<c> <Rn>,<Rm>




```
n = UInt(Rdn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

エンコード T2 ARMv6T2、ARMv7

TST<c>.W <Rn>,<Rm>{,<shift>}




```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
TST<c><q>  <Rn>, <Rm> {,<shift>}
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn>　第一オペランドを格納しているレジスタを指定します。

<Rm>　オプションとしてシフトされ、2 番目のオペランドとして使用されるレジスタを指定します。

<shift>　<Rm> から読み出された値に適用されるシフトを指定します。<shift> を省略した場合、シフトは行われず、両方のエンコードが許可されます。<shift> を指定した場合、エンコード T2 のみが許可されます。どのようなシフトが可能か、およびそれらがどのようにエンコードされるかについては、P.A6-12「*レジスタに適用されるシフト*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

例外

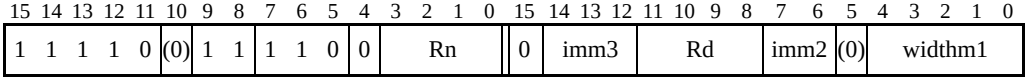
なし

A6.7.143 UBFX

UBFX（符号なしビットフィールドの抽出）は、1 つのレジスタの任意の位置にある、任意の数の隣接するビットを抽出して 32 ビットにゼロ拡張し、結果をデスティネーションレジスタに書き込みます。

エンコード T1 ARMv6T2、ARMv7

UBFX<c> <Rd>,<Rn>,#<lsb>,#<width>




```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

アセンブラ構文

UBFX<c><q> <Rd>, <Rn>, #<lsb>, #<width>

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<Rn> 第一オペランドを格納しているレジスタを指定します。

<lsb> ビットフィールド内の最下位ビットのビット番号で、範囲は 0 ～ 31 です。これによって、lsbit に必要な値が決定されます。

<width> ビットフィールドの幅で、範囲は 1 ～ 32-<lsb> です。widthminus1 に必要な値は <width>-1 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = ZeroExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```

例外

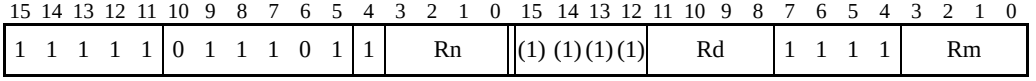
なし

A6.7.144 UDIV

UDIV（符号なし除算）は、32 ビットの符号なし整数レジスタの値を、32 ビットの符号なし整数レジスタの値で除算して、結果をデスティネーションレジスタに書き込みます。条件コードフラグは影響を受けません。

エンコード T1　　プロファイル R バージョンの Thumb-2 ISA

UDIV<c> <Rd>,<Rn>,<Rm>




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if BadReg(d) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

UDIV<c><q> {<Rd>,} <Rn>, <Rm>

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　　デスティネーションレジスタを指定します。<Rd> を省略した場合、<Rn> が使用されます。

<Rn>　　被除数を格納しているレジスタを指定します。

<Rm>　　除数を格納しているレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if UInt(R[m]) == 0 then
        if IntegerZeroDivideTrappingEnabled() then
            GenerateIntegerZeroDivide();
        else
            result = 0;
    else
        result = RoundTowardsZero(UInt(R[n]) / UInt(R[m]));
    R[d] = result<31:0>;
```

例外

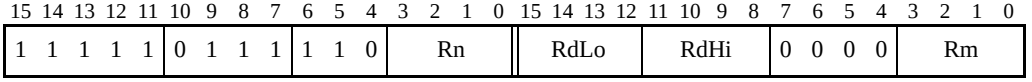
UsageFault

A6.7.145 UMLAL

UMLAL（符号なしロング積和）は、2 つの符号なし 32 ビット値を乗算して 64 ビット値を生成し、別の 64 ビット値と加算します。

エンコード T1 ARMv6T2、ARMv7

UMLAL<c> <RdLo>,<RdHi>,<Rn>,<Rm>




```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

アセンブラ構文

UMLAL<c><q> <RdLo>, <RdHi>, <Rn>, <Rm>

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<RdLo> 加算される値の下位 32 ビットで、同時に結果の下位 32 ビットのデスティネーションレジスタとなります。

<RdHi> 加算される値の上位 32 ビットで、同時に結果の上位 32 ビットのデスティネーションレジスタとなります。

<Rn> 第一オペランドを格納しているレジスタを指定します。

<Rm> 2 番目のオペランドを格納しているレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

例外

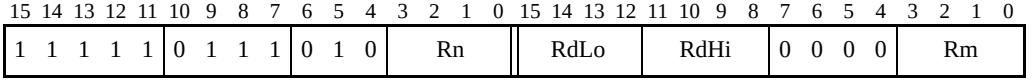
なし

A6.7.146 UMULL

UMULL（符号なしロング乗算）は、2 つの符号なし 32 ビット値を乗算して、64 ビットの結果を生成します。

エンコード T1　　ARMv6T2、ARMv7

UMULL<c> <RdLo>,<RdHi>,<Rn>,<Rm>




```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if BadReg(dLo) || BadReg(dHi) || BadReg(n) || BadReg(m) then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

アセンブラ構文

UMULL<c><q> <RdLo>, <RdHi>, <Rn>, <Rm>

各項目の説明については以下を参照して下さい。

<c><q>　　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<RdLo>　　結果の下位 32 ビットが保存されます。

<RdHi>　　結果の上位 32 ビットが保存されます。

<Rn>　　第一オペランドを格納しているレジスタを指定します。

<Rm>　　2 番目のオペランドを格納しているレジスタを指定します。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

例外

なし

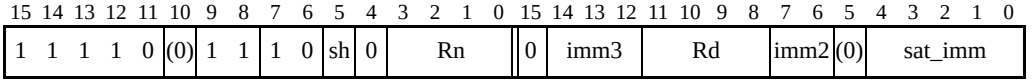
A6.7.147 USAT

USAT（符号なし飽和）は、オプションでシフトされる符号付きの値を、選択された符号なし範囲に飽和させます。

演算が飽和した場合は Q フラグがセットされます。

エンコード T1 ARMv6T2、ARMv7

USAT<c> <Rd>,#<imm5>,<Rn>{,<shift>}




```
if sh == '1' && (imm3:imm2) == '00000' then SEE USAT16;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if BadReg(d) || BadReg(n) then UNPREDICTABLE;
```

アセンブラ構文

```
USAT<c><q>  <Rd>, #<imm>, <Rn> {,<shift>}
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。

<imm>　飽和のビット位置を 0 ～ 31 の範囲で指定します。

<Rn>　飽和させる値を含むレジスタを指定します。

<shift>　オプションのシフトのビット数を指定します。<shift> を省略した場合、LSL #0 が使用されます。
指定する場合は、次のいずれかにする必要があります。
LSL #N　N に許容される値の範囲は 0 ～ 31 です。
ASR #N　N に許容される値の範囲は 1 ～ 31 です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, APSR.C);  // APSR.C ignored
    (result, sat) = UnsignedSatQ(SInt(operand), saturate_to);
    R[d] = ZeroExtend(result, 32);
    if sat then
        APSR.Q = '1';
```

例外

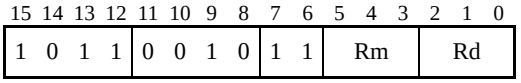
なし

A6.7.148 UXTB

UXTB（符号なしバイト拡張）は、レジスタから 8 ビット値を抽出して、32 ビットにゼロ拡張し、結果をデスティネーションレジスタに書き込みます。8 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6 またはそれ以降すべてのバージョンの Thumb ISA

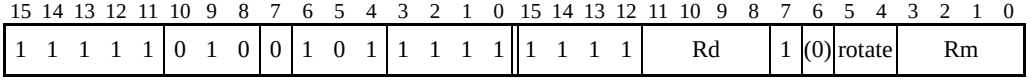
UXTB<c> <Rd>,<Rm>




```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

エンコード T2 ARMv6T2、ARMv7

UXTB<c>.W <Rd>,<Rm>{,<rotation>}




```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
UXTB<c><q>  <Rd>, <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。

<Rm>　2 番目のオペランドを格納しているレジスタを指定します。

<rotation>

次のいずれかのローテーションを指定できます。

- ROR #8.
- ROR #16.
- ROR #24.
- 省略

注

お使いのアセンブラで #0 によるシフトが記述でき、それがシフトなしまたは LSL #0 と等価として処理される場合、このオプションの ROR #0 も記述できるはずです。これは、<rotation> を省略した場合と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```

例外

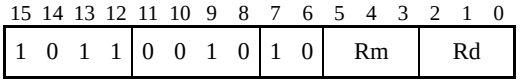
なし

A6.7.149 UXTH

UXTH（符号なしハーフワード拡張）は、レジスタから 16 ビット値を抽出して、32 ビットにゼロ拡張し、結果をデスティネーションレジスタに書き込みます。16 ビット値を抽出する前に 0、8、16、または 24 ビットのローテーションを指定することができます。

エンコード T1 ARMv6 またはそれ以降のすべてのバージョンの Thumb ISA

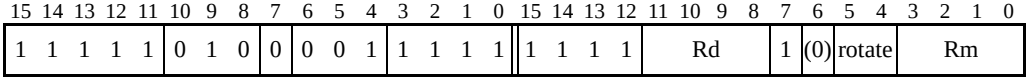
UXTH<c> <Rd>,<Rm>




```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

エンコード T2 ARMv6T2、ARMv7

UXTH<c>.W <Rd>,<Rm>{,<rotation>}




```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if BadReg(d) || BadReg(m) then UNPREDICTABLE;
```

アセンブラ構文

```
UXTH<c><q>  <Rd>, <Rm> {, <rotation>}
```

各項目の説明については以下を参照して下さい。

<c><q>　P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd>　デスティネーションレジスタを指定します。

<Rm>　2 番目のオペランドを格納しているレジスタを指定します。

<rotation>

次のいずれかのローテーションを指定できます。

- ROR #8.
- ROR #16.
- ROR #24.
- 省略

注

お使いのアセンブラで #0 によるシフトが記述でき、それがシフトなしまたは LSL #0 と等価として処理される場合、このオプションの ROR #0 も記述できるはずです。これは、<rotation> を省略した場合と等価です。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```

例外

なし

A6.7.150 WFE

WFE（イベント待ち）はヒント命令です。イベントレジスタがクリアされている場合は、リセット、例外、またはその他のイベントが発生するまで、ソフトウェアを復元する必要のない高速ウェークアップが可能な最小電力状態で、実行を停止します。詳細については、P.AppxD-30「*WaitForEvent()*」を参照して下さい。

全般的なヒント動作については、P.A6-16「*NOP 互換ヒント*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

WFE<c>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0

```
// No additional decoding required
```

エンコード T2 ARMv6T2、ARMv7

WFE<c>.W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	0

```
// No additional decoding required
```

アセンブラ構文

```
WFE<c><q>
```

各項目の説明については以下を参照して下さい。

<c><q> P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if EventRegistered() then
        ClearEventRegister();
    else
        WaitForEvent();
```

例外

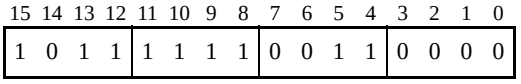
なし

A6.7.151 WFI

WFI（割り込み待ち）はヒント命令です。リセット、非同期例外、またはその他のイベントが発生するまで、ソフトウェアを復元する必要のない高速ウェークアップが可能な最小電力状態で、実行を停止します。詳細については、P.AppxD-31「*WaitForInterrupt()*」を参照して下さい。
全般的なヒント動作については、P.A6-16「*NOP 互換ヒント*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

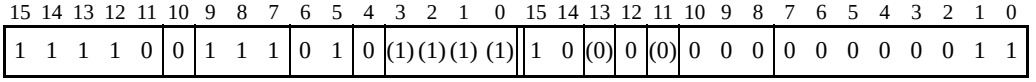
WFI<c>




```
// No additional decoding required
```

エンコード T2 ARMv6T2、ARMv7

WFI<c>.W




```
// No additional decoding required
```

アセンブラ構文

WFI<c><q>

各項目の説明については以下を参照して下さい。

‹c›‹q› P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    WaitForInterrupt();
```

例外

なし

注

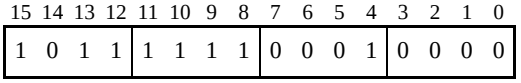
PRIMASK PRIMASK がセットされていて FAULTMASK がクリアされている場合は、任意のアクティブな例外よりもグループ優先度が高く、BASEPRI よりもグループ優先度が高い非同期例外によって、WFI 命令が終了します。例外のグループ優先度が実行グループ優先度以下の場合は、その例外は無視されます。

A6.7.152 YIELD

YIELD（イールド）はヒント命令です。マルチスレッド機能を持つソフトウェアからハードウェアに対して、スピンロックなど、システム全体の性能向上のためにスワップアウト可能なタスクを実行中であることを通知することができます。ハードウェアは、このヒントを使用して、複数のコードスレッドを中断および再開させることができます(この機能をサポートしている場合)。全般的なヒント動作については、P.A6-16「*NOP 互換ヒント*」を参照して下さい。

エンコード T1 ARMv6T2、ARMv7

YIELD<c>

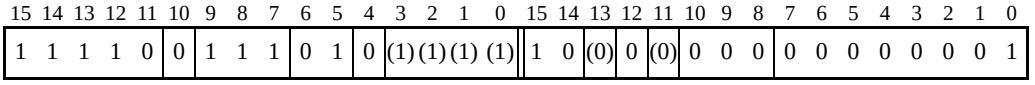


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0

```
// No additional decoding required
```

エンコード T2 ARMv6T2、ARMv7

YIELD<c>.W



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	1

```
// No additional decoding required
```

アセンブラ構文

YIELD<c><q>

各項目の説明については以下を参照して下さい。

‹c›‹q› P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

動作

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
```

例外

なし

パート B

システムレベルのアーキテクチャ

第 B1 章
システムレベル プログラマモデル

本章では、ARMv7-M システムのプログラマモデルの概要を紹介します。本章は、デバイスのテクニカルリファレンス マニュアル (TRM) と同時に読むことを意図して作成されています。TRM には、レジスタインタフェースやプログラミングモデルなど、デバイスに固有の詳細が記載されています。本章は以下のセクションから構成されています。

B1.1 システムレベルの概要

ARM アーキテクチャは階層的に定義されています。第 A2 章 *「アプリケーションレベルのプログラマモデル」* では、基礎となるシステムサポートの上位の、アプリケーションレベルで機能が記述されています。利用可能な機能と、そのサポート方法はアーキテクチャプロファイルに定義されており、そのシステムレベル サポートプロファイル固有のものです。廃止された機能については、本書の付録に記載されています。詳細については、P.AppxC-1 を参照して下さい。

P.A2-13「*特権実行*」で説明されているように、プログラムは特権モードまたは非特権モードで実行できます。システムレベルのサポートではアクセス許可の構成やリソースの制御を行うため、特権アクセスが必要です。これは一般に、オペレーティングシステムによりサポートされます。オペレーティングシステムはアプリケーションへのシステムサービスを、透過的にまたはアプリケーションから開始されたサービス呼び出しによって提供します。また、オペレーティングシステムは割り込みや他のシステムイベントを処理します。このため、システムレベルのプログラマモデルでは、例外が主要な構成要素となります。

さらに、ARMv7-M は従来のアーキテクチャから大きな革新が行われ、従来よりも低いコストでパフォーマンスに優れた ARM アーキテクチャを実現するとともに、ARMv7-R および広範なエンベデッドプロセッサへの強力な移行パスが提供されています。

注

エンベデッドシステム、特に低コストで性能が要求される分野では、オペレーティングシステムとアプリケーションの区別はあいまいになり、両方のソフトウェアが同種のコードベースとして開発されることがあります。

B1.2 システム プログラマモデル

ARMv7-M はメモリマップドアーキテクチャで、物理アドレスとプロセッサレジスタの両方がアーキテクチャにより割り当てられ、イベントのエントリポイント（ベクタ）、システム制御、構成を提供しています。例外ハンドラのエントリポイントは、アドレスポインタのテーブルとして保持されています。

アドレス空間 0xE0000000 ～ 0xFFFFFFFF は、システムレベルで使用するため予約されています。システムアドレス空間の最初の 1MB（0xE0000000 ～ 0xE00FFFFF）は ARM により予約されており、専用ペリフェラルバス (PPB) と呼ばれます。このアドレス空間の残り（0xE0100000 以後）は実装定義で、メモリ属性にいくつかの制約が適用されます。詳細については、P.B2-2「*システム アドレスマップ*」を参照して下さい。

PPB アドレス空間内で、0xE000E000 ～ 0xE000EFFF の 4kB ブロックはシステム制御に割り当てられており、システム制御空間 (SCS) と呼ばれます。SCS は次の機能をサポートしています。

- CPU ID レジスタ
- 一般的な制御と構成（ベクタテーブルのベースアドレスを含む）
- システムハンドラのサポート（システム割り込みおよび例外用）
- SysTick システムタイマ
- ネスト型ベクタ割り込みコントローラ (NVIC) 。このコントローラは、最大 496 の分離した外部割込みをサポートします。すべての例外と割り込みは、共通の優先度付けモデルを使用します。
- フォルトステータスおよび制御レジスタ
- 保護メモリシステム アーキテクチャ (PMSAv7)
- プロセッサデバッグ

詳細については、P.B2-6「*システム制御空間 (SCS)*」を参照して下さい。

B1.2.1 システムレベルの動作と用語の概要

システムレベル アーキテクチャサポートを理解するためには、いくつかの概念が重要です。

モード、特権、スタック

モード、特権、スタックポインタは、ARMv7-M で使用される主要な概念です。

モード　マイクロコントローラのプロファイルは、2 つのモード（スレッドモードとハンドラモード）をサポートしています。ハンドラモードは、例外の結果として開始されます。例外からの復帰は、ハンドラモードでのみ発行できます。

リセットでスレッドモードに入り、例外の復帰の結果としてスレッドモードに入ることもできます。

特権　コードは、特権または非特権として実行できます。非特権実行では、一部のリソースへのアクセスが制限または禁止されます。特権実行では、すべてのリソースにアクセスできます。ハンドラモードは常時、特権状態です。スレッドモードは、特権または非特権のいずれかで実行できます。

スタックポインタ　メイン スタックポインタとプロセス スタックポインタという 2 つの独立したバンク スタックポインタが存在します。メイン スタックポインタは、スレッドモードとハンドラモードのどちらでも使用できます。プロセス スタックポインタは、スレッドモードでのみ使用できます。詳細については、P.B1-7「*SP レジスタ*」を参照して下さい。

モード、特権、使用されるスタックポインタの関係を表 B1-1 に示します。

表 B1-1 モード、特権、スタックポインタの関係

モード	特権	スタックポインタ	（一般的な）使用モデルの例
ハンドラ	特権	メイン	例外処理
ハンドラ	非特権	任意	この組み合わせは予約されています（ハンドラは常に特権モードです）。
ハンドラ	任意	プロセス	この組み合わせは予約されています（ハンドラは常にメインスタックを使用します）。
スレッド	特権	メイン	特権アクセスのみをサポートするシステムで、特権プロセス / スレッドが共通のスタックを使用して実行されます。
スレッド	特権	プロセス	特権アクセスのみをサポートするシステム、または特権スレッドと非特権スレッドが混在するシステムで、プロセス / スレッド用に予約されているスタックを使用して特権プロセス / スレッドが実行されます。
スレッド	非特権	メイン	特権および非特権（ユーザ）アクセスをサポートするシステムで、非特権プロセス / スレッドが共通のスタックを使用して実行されます。
スレッド	非特権	プロセス	特権および非特権（ユーザ）アクセスをサポートするシステムで、非特権プロセス / スレッドがそのプロセス / スレッド用に予約されたスタックを使用して実行されます。

例外

例外は、プログラムの通常のフロー制御を変更するひとつの条件です。例外の動作は、次の 2 つの部分に分けられます。

- 例外の認識：例外イベントが生成され、プロセッサに送られます。
- 例外の処理（起動）：プロセッサが例外の開始、例外からの復帰、例外ハンドラのコードシーケンスを実行します。例外の認識から処理への移行は、即座である可能性があります。

例外は 4 つのカテゴリに分けられます。

リセット　リセットは特別な形式の例外で、リセットがアサートされた場合、現在の実行が回復不能となる可能性がある方法で終了されます。リセットのアサートが解除されると、実行はある決まった位置から再開されます。

スーパバイザコール (SVCall)

SVC 命令によって明示的に引き起こされる例外。スーパバイザコールは、アプリケーションのコードから、ベースとなるオペレーティングシステムに対してシステム（サービス）呼び出しを行うために使用されます。SVC 命令によって、アプリケーションはシステムへの特権アクセスを必要とするシステムコールを発行できます。この命令は、そのアプリケーションのプログラム順序で実行されます。ARMv7-M では、割り込み駆動のサービスコール機構である PendSV もサポートしています。詳細については、P.B1-12「*サポートされている例外の概要*」の*割り込み*を参照して下さい。

フォルト　フォルトは、命令の実行によってエラー条件が発生したことによる例外です。フォルトは、原因となった命令と同期して、または非同期に報告されます。一般には、フォルトは同期して報告されます。不正確 BusFault は、ARMv7-M プロファイルでサポートされている非同期フォルトです。

同期フォルトは常に、フォルトの原因となった命令とともに報告されます。非同期フォルトでは、フォルトの原因となった命令に対してどのように報告されるかは保証されていません。

同期デバッグモニタの例外は、フォルトに分類されます。ウォッチポイントは非同期で、割り込みとして扱われます。

割り込み　リセット、フォルト、スーパバイザコール以外の例外が割り込みです。割り込みはすべて、命令ストリームに対して非同期です。割り込みは一般に、システム内の別の要素がプロセッサと通信を行うために使用されます。これには、他のプロセッサで実行されているソフトウェアも含まれます。

それぞれの例外には、次の属性があります。

- 優先度レベル
- 例外番号
- その例外を取得したときに実行されるエントリポイント（アドレス）を定義する、メモリ内のベクタ。割り込みに関連付けられているコードは、例外ハンドラまたは割り込み処理ルーチン (ISR) と呼ばれます。

リセットを除くそれぞれの例外には、3 つの可能な状態があります。

- 非アクティブの例外は、保留中でもアクティブでもないものです。
- 保留中の例外は、例外イベントが生成され、プロセッサで処理が開始されていないものです。
- アクティブな例外は、プロセッサでハンドラが開始され、処理が完了していないものです。アクティブな例外は、実行中か、より優先度が高い例外によって横取りされているかのいずれかの状態です。

非同期例外はこれら 3 つの状態の他に、同時に保留中かつアクティブになることがあります。こ

の場合、その例外のインスタンスのひとつはアクティブで、2 番目のインスタンスは保留中です。

優先度レベルと例外の横取り

すべての例外には、優先度レベルとして例外優先度が割り当てられています。3 つの例外は値が固定ですが、他のすべての例外は特権ソフトウェアによって変更が可能です。さらに、プロセッサで実行されている命令ストリームには、優先度レベルとして実行優先度がつけられています。例外優先度が実行優先度よりも十分に[1] 高い例外はアクティブになります。この場合、現在実行されている命令ストリームは横取りされ、処理をする例外がアクティブになります。

命令ストリームがリセット以外の例外によって横取りされるときは、主要なコンテキスト情報は自動的にスタックに保存されます。アクティブになった例外ベクタによって指し示されるコードに実行が分岐します。

例外からの復帰

ハンドラモードでは、例外ハンドラは復帰できます。例外がアクティブと保留中の両方である場合（例外の処理中に、その例外の 2 番目のインスタンスが発生した場合）、優先度付けの規則に従って例外ハンドラへの再エントリが行われるか、保留中になります。例外がアクティブのみである場合、非アクティブになります。スタックに保存されている主要な情報が復元され、実行は例外によって横取りされたコードに戻ります。例外からの復帰のターゲットは、例外の開始時にリンクレジスタに保存される値である例外復帰リンクによって決定されます。

実行状態

ARMv7-M は 16 ビットおよび 32 ビットの Thumb 命令のみを実行し、常時 Thumb 状態で実行されます。Thumb 状態は、アーキテクチャの実行ステータスビット (EPSR.T == 1) で示されます。詳細については P.B1-7「*専用プログラム ステータスレジスタ (xPSR)*」を参照して下さい。ARMv7-M で EPSR.T を 0 に設定すると、次の命令の実行時にフォルトが発生します。

デバッグ状態

デバッグイベントで停止するようにコアが構成されていて、デバッグイベントが発生すると、デバッグ状態に入ります。詳細については、第 C1 章 *「ARMv7-M デバッグ」*を参照して下さい。もう 1 つのデバッグ機構（DebugMonitor 例外を生成する）では、デバッグ状態は使用されません。

B1.2.2 レジスタ

ARMv7-M プロファイルには、コアと密結合された次のレジスタが存在します。

- 汎用レジスタ R0 ～ R12
- SP_main および SP_process の 2 つのスタックポインタ レジスタ（R13 のバンクバージョン）
- リンクレジスタ LR (R14)
- プログラムカウンタ (PC)
- フラグ、例外 / 割り込みレベル、実行状態ビットを含むステータスレジスタ

1. 十分に高いという概念は、例外の優先度付けモデルに含まれている優先度のグループ化に関係しています。

- 例外および割り込みの優先度付け方式の管理に対応したマスクレジスタ
- 現在のスタックと特権レベルを識別する制御レジスタ (CONTROL)

この仕様に記載されている他のすべてのレジスタは、メモリマップされています。

SP レジスタ

ARMv7-M では 2 つのスタックがサポートされており、それぞれに独自の（バンク）スタックポインタ レジスタが存在します。

- メインスタック -SP_main
- プロセススタック -SP_process

ARMv7-M 実装では、ビット [1:0] は RAZ/WI として扱われます。ソフトウェアは、ARMv7 プロファイル全体で最大の移植性を実現するため、ビット [1:0] を SBZP として扱う必要があります。SP を明示的に参照する命令によって使用される SP は、P.B1-10「*Thumb 命令セットで ARM コアレジスタへのアクセスを行う擬似コードの詳細*」で説明されている関数 LookUpSP() に従って選択されます。

リセット時には SP_main が選択され、初期化されます。

専用プログラム ステータスレジスタ (xPSR)

システムレベルのプロセッサステータスは、次の 3 つのカテゴリに分けられます。これらのステータスは、MRS および MSR 命令を使用して、個別のレジスタ、3 つのうちの任意の 2 つの組み合わせ、または 3 つすべての組み合わせとしてアクセスできます。

- アプリケーションプログラム ステータスレジスタ (APSR) - ユーザが書き込み可能なフラグ。MSR および MRS 命令による APSR のユーザ書き込み可能なフラグに対する処理は、すべての ARMv7 プロファイルを通して一貫しています。
- 割り込みプログラム ステータスレジスタ (IPSR) - 例外番号（現在実行されているもの）
- 実行プログラム ステータスレジスタ (EPSR) - 実行状態ビット

APSR、IPSR、EPSR の各レジスタは、32 ビットレジスタ内の相互に排他なビットフィールドとして割り当てられています。APSR、IPSR、EPSR レジスタの組み合わせを xPSR レジスタと呼びます。

表 B1-2 xPSF レジスタのレイアウト

	31	30	29	28	27	26	25	24	23 … 16	15 … 10	9	8	… 0
APSR	N	Z	C	V	Q								
IPSR													0 または例外番号
EPSR						ICI/IT	T			ICI/IT	a		

a. EPSR[9] は予約ですが、xPSR のコンテキスト情報をスタックに保存するためのメモリでこのビットに関連付けられている場所は、スタックアライメント サポートに割り当てられています。詳細については、P.B1-16「*例外開始時のスタックアライメント*」を参照して下さい。

APSR はフラグをセットする命令によって変更され、IT での条件付き実行と条件付き分岐命令の評価に使用されます。フラグ (NZCVQ) については、P.A2-11「*ARM コアレジスタ*」を参照して下さい。これらのフラグのリセット時の値は予測不能です。

IPSR は、例外の開始時と終了時に書き込まれ、MRS 命令を使用して読み出すことができます。MSR 命令を使用して IPSR に書き込んだ場合、その書き込みは無視されます。IPSR の例外番号フィールドはリセット時にクリアされ、次のように定義されます。

- スレッドモードでは、この値は 0 です。
- ハンドラモードでは、この値は P.B1-13「*例外番号*」に定義されている例外番号を反映しています。

EPSR には、IT 命令、または割り込み継続ロード / ストア命令をサポートするため、T ビットおよびオーバーレイされた IT/ICI 実行状態ビットが含まれています。これらのフィールドはすべて、MRS 命令を使用すると 0 として読み出されます。MSR による書き込みは無視されます。

EPSR の T ビットは ARM アーキテクチャのインターワーキングモデルをサポートしていますが、ARMv7-M は Thumb 命令の実行のみをサポートしているため、このビットの値は常に T ビット == 1 です。Thumb 命令のインターワーキング規則に準拠した PC の更新では、それに応じて T ビットを更新する必要があります。EPSR の T ビットがクリアの状態で命令を実行すると、無効状態 (INVSTATE) UsageFault が発生します。リセット時には、T ビットがセットされ、IT/ICI ビットがクリアされます（詳細については、P.B1-15「*リセット*」を参照して下さい）。

ICI/IT ビットは IT 状態の保存、または例外の後で続行可能な命令の状態の保存に使用されます。

- IT ビットは、命令シーケンスにおける条件付き実行のコンテキスト情報を提供し、命令の割り込みと再開が適切な時点で行われるようにします。詳細については、第 A6 章 *「Thumb 命令の詳細」* にある IT 命令の定義を参照して下さい。
- ICI ビットは、例外の後で続行可能なマルチサイクルのロード / ストア命令について、残っているレジスタリスト情報を提供します。

IT/ICI ビットは、表 B1-3 に示すように割り当てられています。

表 B1-3 EPSR の ICI/IT ビットの割り当て

EPSR[26:25]	EPSR[15:12]	EPSR[11:10]	追加情報
IT[1:0]	IT[7:4]	IT[3:2]	詳細については、P.A6-10 「*ITSTATE*」を参照して下さい。
ICI[7:6] (’00’)	ICI[5:2] (reg_num)	ICI[1:0] (’00’)	<reg_num> は、続行で使用されるレジスタリスト内の開始位置です。

例外の後で続行可能な命令が IT 構造内で使用された場合、IT 機能が ICI 機能よりも優先されます。この条件では、複数サイクルのロードまたはストア命令は再始動可能として扱われます。

個別または組み合わされたレジスタの使用されていないビットは、すべて予約されています。

専用マスクレジスタ

例外の優先度管理のための専用マスクレジスタは 3 つあります。

- 例外マスクレジスタ PRIMASK は 1 ビットの値です。
- ベース優先度マスク BASEPRI は 8 ビットの値です。
- フォルトマスク FAULTMASK は 1 ビットの値です。

すべてのレジスタは、リセット時にクリアされます。非特権での書き込みはすべて無視されます。このレジスタのフォーマットを表 B1-4 に示します。

表 B1-4 専用マスクレジスタ

	31 … 8	7 … 1	0
PRIMASK	予約		PM
FAULTMASK	予約		FM
BASEPRI	予約	BASEPRI	

これらのレジスタには、MSR または MRS 命令を使用してアクセスできます。CPS 命令は、PRIMASK および FAULTMASK の変更に使用できます。

詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

専用制御レジスタ

専用 CONTROL レジスタは 2 ビットのレジスタで、次のように定義されています。

- ビット [0] は、スレッドモードの特権を定義します（ハンドラモードは常に特権モードです）。
- ビット [1] は、スレッドモードのスタックを定義します（ハンドラモードは常に SP_main が使用されます）。
- ビット [31:2] は予約されています。

CONTROL レジスタは、リセット時にクリアされます。MRS 命令を使用してレジスタを読み出すことができ、MSR 命令を使用してレジスタに書き込むことができます。非特権の書き込みアクセスは無視されます。

CONTROL レジスタへの書き込みアクセスが、次の命令の実行前に有効になることを保証するた

めには ISB バリア命令が必要です。詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

専用レジスタの予約ビット

専用レジスタの未使用ビットはすべて予約されています。予約ビットにアクセスする MRS および MSR 命令は、これらのビットを RAZ/WI として扱います。将来のソフトウェア互換性を維持するため、これらのビットは UNK/SBZP です。新しいプロセスでこのレジスタを初期化するときには 0 を書き込みます。それ以外の場合にソフトウェアで専用レジスタの更新または復元を行うときは、予約ビットについては元の値を復元する必要があります。

専用レジスタの更新とメモリオーダモデル

CONTROL レジスタへの書き込みを除いて、CPS または MSR 命令による専用レジスタの変更では、常に次の条件が保証されます。

- これらの命令や、プログラム順序で前にある命令に影響しない。
- プログラム順序で、これらの変更よりも後にあるすべての命令から可視である。

Thumb 命令セットで ARM コアレジスタへのアクセスを行う擬似コードの詳細

次の擬似コードは、P.A6-17「*ARMv7-M Thumb 命令のアルファベット順リスト*」で定義されている Thumb 命令セットの動作について、汎用レジスタへのアクセスをサポートします。

```
// The M-profile execution modes.

enumeration Mode {Thread, Handler};

// The names of the core registers. SP is a banked register.

enumeration RName {RName0, RName1, RName2, RName3, RName4, RName5, RName6,
                   RName7, RName8, RName9, RName10, RName11, RName12,
                   RNameSP_main, RNameSP_process, RName_LR, RName_PC};

// The physical array of core registers.
//
// _R[RName_PC] is defined to be the address of the current instruction.
// The offset of 4 bytes is applied to it by the register access functions.

array bits(32) _R[RName];

// LookUpSP()
// ==========

RName LookUpSP()
    RName SP;
    if CONTROL<1> == 1
        if Mode==Thread then
            SP is RNameSP_process;
        else
```



```
            UNPREDICTABLE;
    else
        SP is RNameSP_main;
    return SP;


// R[] - non-assignment form
// =========================

bits(32) R[integer n]
    assert n >= 0 && n <= 15;
    if n == 15 then
        result = _R[RName_PC] + 4;
    elsif n == 14 then
        result = _R[RName_LR]
    elsif n == 13 then
        LookUpSP();
        result = _R[SP];
    else
        result = _R[RName:n];
    return result;


// R[] - assignment form
// =====================

R[integer n] = bits(32) value
    assert n >= 0 && n <= 14;
    if n == 13 then
        LookUpSP();
        _R[SP] = value;
    else
        _R[RName:n] = value;
    return;



// BranchTo()
// ==========

BranchTo(bits(32) address)
    _R[RName_PC] = address;
    return;
```

B1.2.3 例外モデル

例外モデルは、ARMv7-M プロファイルのアーキテクチャとシステム正当性の中心となるものです。ARMv7-M プロファイルが他の ARMv7 プロファイルと異なる点は、例外の開始時と終了時に主要なコンテキスト状態をハードウェアで保存および復元することと、例外のエントリポイントをベクタテーブルで判断することです。また、ARMv7-M プロファイルでの例外のカテゴリ分けは、他の ARMv7 プロファイルとは異なっています。

サポートされている例外の概要

ARMv7-M プロファイルでは、次の例外がサポートされています。

リセット　ARMv7-M プロファイルでは、2 レベルのリセットがサポートされ、システム状態の異なるレベルのリセットができます。リセットのレベルによって、リセットがアサート解除されたときにリセット時の値に強制的に設定されるレジスタが異なります。

- パワーオンリセット (POR) は、コア、システム制御空間、デバッグロジックをリセットします。
- ローカルリセットは、コアと、デバッグに関連する一部のリソースを除くシステム制御空間をリセットします。

リセット例外は常時有効で、優先度は - 3 に固定されています。

NMI- マスク不能割り込み　マスク不能割り込みは、リセットを除いて最も優先度が高い例外です。この割り込みは常時有効で、優先度は - 2 に固定されています。

HardFault　HardFault は他の例外機構によって処理できないすべてのクラスの例外のために存在する汎用のフォルトです。HardFault は通常、回復不能なシステム障害の状態で使用されますが、これは必須ではなく、HardFault の使用法の一部では回復が可能です。HardFault は常時有効で、優先度は - 1 に固定されています。

HardFault はフォルトの昇格に使用されます。

MemManage　MemManage フォルトはメモリ保護に関連するフォルトを処理し、メモリ保護ユニットまたは決められたメモリ保護の制約によって判定されます。このフォルトは命令とデータのいずれによって生成されたメモリトランザクションでも発生します。このフォルトは無効にできます（その場合、MemManage フォルトは HardFault に昇格されます）。MemManage の優先度は構成可能です。

BusFault　BusFault フォルトは、命令とデータのいずれかで発生したメモリトランザクションで起きたメモリ関連のフォルトのうち、MemManage フォルトで処理されないものを処理します。これらのフォルトは一般に、システムバスで検出されたエラーによって発生します。実装者は、例外をトリガした条件に応じて、BusFault を同期または非同期に報告できます。このフォルトは無効にできます（その場合、同期 BusFault は HardFault に昇格されます）。BusFault の優先度は構成可能です。

UsageFault　UsageFault は、命令の実行により発生した、メモリ関連ではないフォルトを処理します。UsageFault は、次のような多くの状況で引き起こされます。

- 未定義命令
- 命令実行時の無効な状態
- 例外からの復帰時のエラー
- 無効または使用禁止のコプロセッサへのアクセス

次の状況は、コアがこれらの状況を報告するように構成されている場合に UsageFault を引き起こします。

- ワードまたはハーフワードのメモリアクセスにおける、アンアラインドアドレス
- 0 による除算

UsageFault は無効にできます（その場合、UsageFault は HardFault に昇格されます）。UsageFault の優先度は構成可能です。

デバッグモニタ　一般に、DebugMonitor 例外は同期例外で、フォルトに分類されます。ウォッチポイントは非同期で、割り込みとして動作します。デバッグモニタ例外は、ホールトデバッグが無効で、デバッグモニタ サポートが有効な場合に発生します。DebugMonitor の優先度は構成可能です。

SVCall　このスーパバイザコールは、SVC 命令によって発生した例外を処理します。SVCall は常に有効で、優先度は構成可能です。

割り込み　ARMv7-M プロファイルは、2 つのシステムレベル割り込み（ソフトウェアで非同期のシステムコールを生成するための PendSV と、ARMv7-M プロファイルにタイマを統合するための SysTick）、および最大 496 の外部割り込みをサポートしています。すべての割り込みの優先度は構成可能です。

PendSV[1] は常時有効な割り込みです。SysTick を無効にすることはできません。

注

ハードウェアで生成される SysTick イベントは抑制できますが、ICSR.PENDSTSET および ICSR.PENDSTCLR は常にソフトウェアから使用可能です。

他のすべての割り込みは無効にできます。割り込みはソフトウェアで保留状態にセット、または保留状態をクリアでき、PendSV 以外の割り込みはハードウェアで保留状態にセットできます。

例外番号

すべての例外には、表 B1-5 で定義されている例外番号が関連付けられています。

表 B1-5 例外番号

例外番号	例外
1	リセット
2	NMI
3	HardFault
4	MemManage
5	BusFault
6	UsageFault
7-10	予約

1. サービス（システム）コールは、ベースとなるオペレーティングシステムからのサービスを必要とするアプリケーションによって使用されます。PendSV に関連付けられたサービスコールは、割り込みの取得時に実行されます。プログラム実行に関して同期に実行されるサービスコールを行うには、SVC 命令（SVCall 例外）を使用します。

表 B1-5 例外番号

例外番号	例外
11	SVCall
12	デバッグモニタ
13	予約
14	PendSV
15	SysTick
16	外部割込み (0)
…	…
16 + N	外部割込み (N)

ベクタテーブル

ベクタテーブルには、リセット時にスタックポインタを初期化するための値と、すべての例外ハンドラに対応するエントリポイントのアドレスが含まれています。例外ハンドラのエントリに対応するベクタテーブルのエントリ順序は、例外番号（上記参照）により定義されます。この対応を表 B1-6 に示します。

表 B1-6 ベクタテーブルのフォーマット

ワードオフセット	説明 - すべてのポインタアドレスの値
0	SP_main（メイン スタックポインタのリセット時の値）
例外番号	例外番号に対応する例外

リセット時に、ベクタテーブルのベースアドレスは、CODE パーティションの実装定義の値に初期化されます。テーブルの現在の場所は、ベクタテーブル オフセットレジスタ (VTOR) を使用して、ARMv7-M メモリマップの CODE または SRAM パーティション内で判定、または変更できます。詳細については、ARM コアまたはデバイスの仕様を参照して下さい。

例外の優先度と横取り

優先度の決定では、より低い数値が優先されます。つまり、割り当てられている値が低いほど、優先度レベルは高くなります。同じ優先度レベルを割り当てられている例外は、例外番号に従い、アーキテクチャ内で固定された優先順位で選択されます。

リセット、マスク不能割り込み (NMI) 、HardFault は、それぞれ - 3、- 2、- 1 に優先度が固定されています。他のすべての例外は、ソフトウェア制御で優先度を設定でき、リセット時に設定がクリアされます。

すべての例外の優先度は、システム制御空間内のレジスタ（具体的には、システム制御ブロックと NVIC のレジスタ）に設定されています。

複数の例外の優先度番号が同じ場合、最も低い例外番号を持つ保留中の例外が優先されます。例外がアクティブになった後では、より優先度の高い（優先度番号が低い）例外のみが横取りできます。

優先度のグループ化、優先度の昇格、横取りなど優先度サポートの詳細については、ARM コアまたはデバイスの仕様を参照して下さい。

リセット

リセットがアサートされると、現在の実行状態は保存されずに破棄されます。リセットがアサート解除されると、アサートされたリセットにより制御されるすべてのレジスタは、リセット時の値に設定されます。

詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

例外の開始

命令ストリームの横取り時に、コンテキストの状態はハードウェアによって、SP レジスタの 1 つで指し示されるスタックへ保存されます（P.B1-7「*SP レジスタ*」参照）。使用されるスタックは、例外が発生した時点でプロセッサがどのモードにあるかによって決定されます。

スタックに保存されたコンテキストは、ARM アーキテクチャ プロシージャ呼び出し標準 (AAPCS) に対応しています。この対応により、AAPCS 準拠のプロシージャを例外ハンドラとして使用できます。

フル下降スタックフォーマットが使用され、スタックに 32 ビットワードが保存される（コンテキストのプッシュ）直前にスタックポインタがデクリメントされ、スタックから 32 ビットワードが読み出された（コンテキストのポップ）直後にインクリメントされます。次に示す 8 つの 32 ビットワードが、メモリ内のアドレスで降順に保存されます。

xPSR, ReturnAddress() , LR (R14) , R12, R3, R2, R1, and R0

ReturnAddress() は、以下に定義されているように、例外の処理が完了したときに実行が復帰するアドレスです。

```
// ReturnAddress()
// ===============

Bits(32) ReturnAddress() returns the following values based on the exception cause
         // NOTE: ReturnAddress() is always halfword aligned - bit<0> is always zero
         //       xPSR.IT bits saved to the stack are consistent with ReturnAddress()

// Exception Type          Address returned
// ==============          ================

// NMI:                      Address of Next Instruction to be executed
// HardFault (precise):      Address of the Instruction causing fault
// HardFault (imprecise):    Address of Next Instruction to be executed
// MemManage:                Address of the Instruction causing fault
// BusFault (precise):       Address of the Instruction causing fault
// BusFault (imprecise):     Address of Next Instruction to be executed
// UsageFault:               Address of the Instruction causing fault
// SVC:                      Address of the Next Instruction after the SVC
// DebugMonitor (precise):   Address of the Instruction causing fault
// DebugMonitor (imprecise): Address of Next Instruction to be executed
```



```
// IRQ:                    Address of Next Instruction to be executed after an interrupt
```

注

HardFault の優先度に昇格されたフォルトは、元のフォルトの ReturnAddress() 動作を保持します。

IRQ には、SysTick と PendSV が含まれます。

例外開始時のスタックアライメント

ARMv7-M には、すべての例外が 8 バイトのスタックアライメントで開始されることを保証する構成オプションがあります。ARMv7-M のスタックポインタは、最低でも 4 バイトアラインドであることが保証されています。例外は命令の任意の境界で発生の可能性があるため、例外がアクティブになったときに現在のスタックポインタが 8 バイトアラインドでない可能性があります。

AAPCS に準拠した動作を行うには、開始時にスタックポインタが 8 バイトアラインドの必要があります。[1] 例外ハンドラは AAPCS に準拠した関数として作成されることが前提なため、システムは渡されるすべての引数についてスタックの自然なアライメントを保証する必要があります。8 バイトアライメントの条件は、この機能を使用してハードウェアで保証されます。

構成および制御レジスタの STKALIGN ビット（詳細については、関連する ARM コアまたはデバイスの仕様を参照して下さい）は、スタックの 8 バイトアライメント機能を有効にするために使用されます。このビットは、次の場合には実装定義です。

- このビットがソフトウェアでプログラム可能である。実装には、STKALIGN ビットのリセット時の値を決定する、構成入力信号を含めることができます。リセット時の値を決定する構成入力信号がない場合、このビットのリセット時の値は 0 です。
- このビットは RAZ/WI です。STKALIGN は、ARMv7-M アーキテクチャの開発サイクルの後期に追加されたものです。初期の実装は、この機能をサポートしていない可能性があります。

このビットは、システムの起動シーケンスで、8 バイトアライメントの対応が必要になる前にセットする必要があります。

注

NMI 例外は、リセットから起動できます。CCR.STKALIGN ビットがリセット時にクリアされる場合、ソフトウェアで NMI 例外ハンドラが 8 バイトアライメントを必要としないことを保証する必要があります。

4 バイトアラインド スタックのサポート (CCR.STKALIGN == "0") は ARMv7-M では推奨されません。

例外の開始時から復帰時までの間にこのビットがクリアされ、例外の開始時にスタックが 8 バイトアラインドでなかった場合、システム破壊が発生する可能性があります。

1. AAPCS に準拠した関数は、サイズが 1、2、4、8 バイトの基本データについて、自然なアライメントを保持する必要があります。その代わりに、準拠したコードではこのアライメントを前提とすることができます。認証されていない依存性をサポートするため、スタックポインタは一般に、準拠した関数の開始時に 8 バイトアラインドとする必要があります。ベースとなる実行環境から直接関数が開始される場合、その環境ではスタックアライメントの条件に従い、準拠したコードがあらゆる条件で正しく実行されるという認証されていない保証を与える必要があります。

例外からの復帰

例外からの復帰は、ハンドラモードで次のいずれかの命令によって PC に値 0xFFFFFFFX がロードされたときに発生します。

- PC のロードを含む POP/LDM
- PC をデスティネーションとする LDR
- 任意のレジスタとの BX

この方法で使用される場合、PC に書き込まれる値は介在を受けて、EXC_RETURN 値と呼ばれます。

EXC_RETURN[28:4] は特別な条件で予約されており、すべてのビットに 1 を書き込むか、保持する必要があります。すべて 1 以外の値は予測不能です。EXC_RETURN[3:0] は、表 B1-7 に定義されているように復帰情報を示します。

表 B1-7 例外からの復帰動作

EXC_RETURN[3:0]	
0bXXX0	予約
0b0001	ハンドラモードへの復帰 例外からの復帰時には、メインスタックから状態が復元されます。 復帰時の実行はメインスタックを使用します。
0b0011	予約
0b01X1	予約
0b1001	スレッドモードへの復帰 例外からの復帰時には、メインスタックから状態が復元されます。 復帰時の実行はメインスタックを使用します。
0b1101	スレッドモードへの復帰 例外からの復帰時には、プロセススタックから状態が復元されます。 復帰時の実行はプロセススタックを使用します。
0b1X11	予約

例外のステータスと制御

システム制御空間内のシステム制御ブロック（P.B2-7「*システム制御ブロック (SCB)*」）は、例外モデルを管理するために必要なレジスタサポートを提供します。

レジスタの詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

フォルトの動作

ARMv7-M の例外優先度付け方式に従い、正確なフォルト例外のハンドラは次のいずれかの方法で実行されます。

- 指定の例外ハンドラを実行する。
- HardFault 例外を実行する。
- 回復不能フォルトに対応するロックアップ動作を採用する。

詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

注

ARMv7-M プロファイルでは一般に、プロセッサが優先度 - 1 またはそれ以上の優先度で実行されている場合、発生するフォルトまたはスーパバイザコールはすべて致命的で、完全に予期しないものであることが想定されています。

リセットの管理

アプリケーション割り込みおよびリセット制御レジスタは、システムリセット用に 2 つの機構を提供しています。

- 制御ビット SYSRESETREQ は、外部システムリソースによるリセットを要求します。この要求によりリセットされるシステムコンポーネントは実装定義です。SYSRESETREQ は、ローカルリセットを起こす必要があります。
- 制御ビット VECTRESET（デバッグ機能、以下の*リセットとデバッグ*参照）は、ローカルリセットを起こします。この制御の結果として、システムの他の部分がリセットされるかどうかは実装定義です。

  注

  SYSRESETREQ と VECTRESET を同じ書き込みアクセスでセットする（1 を書き込む）ことは避けて下さい。これらのビットに同時に 1 を書き込んだ場合、動作は予測不能です。

SYSRESETREQ については、リセットが直ちに行われる保証はありません。関連する制御ビットへの書き込みの後でリセットを同期するため、一般的に次のようなコードシーケンスを使用します。

```
     DSB;
Loop  B   Loop;
```

詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

リセットとデバッグ

デバッグロジックは、パワーオンリセットにより完全にリセットされます。ローカルリセットでは、部分的にのみリセットされます。詳細については、P.C1-11「*デバッグとリセット*」を参照して下さい。デバッガは、コアが停止しているときには VECTRESET のみを使用する必要があります。これに従わない場合、結果は予測不能です。

電力管理

ARMv7-M プロファイルは、イベントや割り込み待ちのときに実行を中断するため、WFE および WFI ヒントの使用をサポートしています。システム状態が保持される限り、この方法で実行を中断するとき、実装でどのレベルの電力削減を使用するかは実装定義です。WFE および WFI を使用するコードは、デバッグホールトまたは他の実装定義の理由で発生する擬似ウェークアップイベントを処理する必要があります。

アプリケーション割り込みおよびリセット制御レジスタは、制御と構成の機能を提供します。

WFI および WFE 命令の相違点とシステム関連の動作の詳細については、P.A6-276「*WFI*」および P.A6-275「*WFE*」を参照して下さい。

詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

第 B2 章
システム アドレスマップ

ARMv7-M はメモリマップのアーキテクチャです。本章ではシステム アドレスマップについて説明します。本章は、デバイスのテクニカル リファレンスマニュアル (TRM) と同時に読むことを意図して作成されています。TRM には、レジスタの定義やプログラミングモデルなど、デバイスの詳細が記載されています。本章は以下のセクションから構成されています。

B2.1 システム アドレスマップ

ARMv7-M では、32 ビットのアドレス空間に、コード、データ、ペリフェラル用の領域があらかじめ定義されていて、オンチップ（コアに密結合）およびオフチップのリソース用の領域も定義されています。アドレス空間では、次に示す 8 つの 0.5GB の 1 次パーティションがサポートされています。

- コード
- SRAM
- ペリフェラル
- 2 つの RAM 領域
- 2 つのデバイス領域
- システム

物理アドレスは、イベントのエントリポイント（ベクタ）、システム制御、および構成に使用するため、アーキテクチャによって割り当てられています。イベントのエントリポイントは、すべてテーブルベースアドレスをもとに決定され、ベースアドレスはリセット時に実装定義の値に構成され、システムの構成および制御のために予約されているアドレス空間に保持されます。この条件、およびシステムの他の要求に対応するため、アドレス空間の 0xE0000000 〜 0xFFFFFFFF はシステムレベルでの使用に RESERVED されています。

ARMv7-M のデフォルトのアドレスマップを P.B2-3 表 B2-1 に示します。

- XN はその領域が実行不可であることを示し、この領域の実行を試みると、常にフォルト（MemManage 例外）が発生します。
- キャッシュの列は、システムキャッシュをサポートするための内部 / 外部キャッシュポリシーを示しています。このポリシーでは、宣言されたキャッシュのタイプを降格することはできますが、昇格することはできません。

  WT : ライトスルーで、キャッシュなしとして扱うこともできます。

  WBWA : ライトバック、書き込み割り当てで、ライトスルーまたはキャッシュなしとして扱うこともできます。
- 共有可という記述は、アクセスが複数のエージェント、つまり複数のプロセッサ、DMA エージェント、または両方から、コヒーレントなメモリドメイン内で共有をサポートすることを意図していることをシステムに通知します。
- アドレス空間全体のうちどの部分が読み出し / 書き込みで、どの部分が読み出し専用（フラッシュメモリなど）で、どの部分がアクセス不可（アドレスマップで何も割り当てられていない部分）かは IMPLEMENTATION DEFINED です。
- 0.5GB のアドレス境界にまたがる、アンアラインドまたは複数ワードアクセスは予測不能です。

メモリ属性およびメモリモデルの詳細については、第 A3 章 *「ARM アーキテクチャのメモリモデル」* を参照して下さい。

表 B2-1 ARMv7-M のアドレスマップ

開始	名前	デバイスタイプ	XN	キャッシュ	説明
0x00000000 ～ 0x1FFFFFFF	コード	ノーマル	-	WT	フラッシュまたは他のコード。実装により、使用するメモリをこれよりも少なくすることはできますが、この領域はアドレス 0x0 から始まる必要があります。
0x20000000 ～ 0x3FFFFFFF	SRAM	ノーマル	-	WBWA	オンチップ RAM。SRAM はベースアドレスから始まる必要がありますが、他の種類はオフセット可能です。
0x40000000 ～ 0x5FFFFFFF	ペリフェラル	デバイス	XN	-	オンチップペリフェラルのアドレス空間
0x60000000 ～ 0x7FFFFFFF	RAM	ノーマル	-	WBWA	L2/L3 キャッシュサポートについてライトバック、書き込み割り当てのキャッシュ属性を持つメモリ
0x80000000 ～ 0x9FFFFFFF	RAM	ノーマル	-	WT	ライトスルーのキャッシュ属性を持つメモリ
0xA0000000 ～ 0xBFFFFFFF	デバイス	デバイス、共有可	XN	-	共有可能なデバイス空間
0xC0000000 ～ 0xDFFFFFFF	デバイス	デバイス	XN	-	共有不可のデバイス空間
0xE0000000 ～ 0xFFFFFFFF	システム	-	XN	-	PPB およびベンダシステム ペリフェラル用のシステムセグメント
+0000000	PPB	SO、(共有)	XN		専用ペリフェラルバスとして予約されている 1MB の領域。PPB は、システム制御空間およびデバッグ機能を含む主要なリソースをサポートしています。
+0100000	Vendor_SYS	デバイス	XN		ベンダシステム。ベンダリソースは 0xF0000000（+GB オフセット）で開始することが推奨されます。

ユーザ（非特権）およびスーパバイザ（特権）ソフトウェアモデルをサポートするため、アクセス権限を制御するためのメモリ保護方式が必要です。ARMv7-M (PMSAv7) の保護メモリシステムアーキテクチャは、オプションのシステムレベル機能で、P.B2-12「*保護メモリシステム アーキテクチャ*」に説明されています。PMSAv7 の実装の 1 つは、メモリ保護ユニット (MPU) と呼ばれます。

表 B2-1 に示されているアドレスマップは、MPU が禁止されているときのデフォルトのマップで、MPU が存在しないときにはこのアドレスマップのみがサポートされます。MPU が許可されているとき、特権アクセス用のバックグラウンド領域としてデフォルトマップを許可できます。

注

MPU が許可されている場合、MPU がシステム空間（アドレス 0xE0000000 以上）に関連するデフォルトのメモリマップ属性をどのように変更できるかは制約があります。システム空間は、常に XN にマークされています。デフォルトがデバイスに指定されているシステム空間は、ストロングリオーダに変更できますが、ノーマルメモリにマップすることはできません。PPB メモリ属性を MPU によってリマップすることはできません。

B2.1.1 PPB レジスタアクセスに適用される一般的な規則

専用ペリフェラルバス (PPB) 用のアドレス範囲である 0xE0000000 ～ 0xE0100000 は、次のような一般的な規則をサポートしています。

- この領域はストロングリオーダメモリとして定義されています。詳細については、P.A3-26「*ストロングリオーダメモリ*」および P.A3-27「*メモリアクセスの制約事項*」を参照して下さい。
- レジスタは、プロセッサのエンディアン形式に関係なく、常にリトルエンディアンとしてアクセスされます。
- 一般に、レジスタはワードアクセスのみをサポートしています。バイトおよびハーフワードアクセスは UNPREDICTABLE です。いくつかのレジスタ（具体的には優先度およびフォルトステータス レジスタ）はバイトアラインドのビットフィールドが連結したもので、それぞれ異なるリソースに影響します。これらのレジスタ[1] は、32 ビットのレジスタ ベースアドレス内で適切なアドレスオフセットを持つ、8 ビットまたは 16 ビットレジスタとして宣言できます。
- セットするという用語は、値として 1 を書き込むことを意味します。クリアするという用語は、0 を書き込むことを意味します。これらの用語が複数のビットに対して使用される場合、すべてのビットに同じ値が書き込まれます。
- 禁止するという用語は、ビットに値として 0 を書き込むことを、許可するという用語は、ビットに 1 を書き込むことを意味します。
- ビットが読み出し時にクリアとして定義されている場合、そのビットをセットするイベントと同時に読み出されているとき、以下のアトミックな動作が保証される必要があります。
  - ビットが 1 として読み出される場合、ビットはその読み出し動作によってクリアされます。
  - ビットが 0 として読み出される場合、そのビットはセットされ、以後の読み出し操作によって読み出し / クリアされます。
- 予約されているレジスタまたはビットフィールドは、UNK/SBZP として扱う必要があります。

PPB への非特権（ユーザ）アクセスは、特記されていない限り BusFault エラーを引き起こします。特記すべき例外には次のものがあります。

- コンフィギュレーション制御レジスタの制御ビットをプログラムすると、システム制御空間のソフトウェアトリガ割り込みレジスタに対して、非特権アクセスが許可されます。
- デバッグ関連のリソースについては、P.C1-6「*デバッグレジスタへのアクセスに適用され*

1. レジスタは、定義で明確に記述されている場合のみ、バイトおよびハーフワードアクセスをサポートします。

る一般的な規則」に例外の詳細が記載されています。

注

フラッシュパッチおよびブレークポイントブロック（FPB、P.C1-23「*フラッシュパッチとブレークポイント (FPB) のサポート*」参照）は、デバッグリソースに割り当てられています。または、FPB リソースを製品の保守ポリシーの一部として、ソフトウェアを更新する手段に使用することもできます。FPB のアドレスリマップの動作は、デバッグ操作に固有のものではありません。FPB リソースがソフトウェアの保守に割り当てられると、デバッグ機能が低下します。

B2.2 システム制御空間 (SCS)

システム制御空間は、メモリマップされた 4kB のアドレス空間で、専用レジスタとともに、コンフィギュレーション、ステータスレポート、制御用の 32 ビットレジスタ群を構成します。SCS は、次のグループに分割されます。

- システム制御 /ID
- CPUID 空間
- システム制御、コンフィギュレーション、ステータス
- フォルト通知
- SysTick システムタイマ
- ネスト型ベクタ割り込みコントローラ (NVIC)
- 保護メモリシステム アーキテクチャ (PMSAv7) -P.B2-12「*保護メモリシステム アーキテクチャ*」参照
- システムデバッグ - 第 C1 章 *「ARMv7-M デバッグ」*

SCS レジスタグループによるアドレス空間の分割は、表 B2-2 に示すように定義されています。

表 B2-2 SCS アドレス空間の領域

システム制御空間（アドレス範囲 0xE000E000 **〜** 0xE000EFFF**)**		
グループ	**アドレス範囲**	**注**
システム制御 /ID	0xE000E000 〜 0xE000E00F	割り込みコントローラタイプと、補助制御レジスタが含まれます。
	0xE000ED00 〜 0xE000ED8F	システム制御ブロックで、1 次 (CPUID) レジスタが含まれます。
	0xE000EF00 〜 0xE000EFCF	SW トリガ例外レジスタが含まれます。
	0xE000EFD0 〜 0xE000EFFF	マイクロコントローラ固有の ID 空間
SysTick	0xE000E010 〜 0xE000E0FF	システムタイマ
NVIC	0xE000E100 〜 0xE000ECFF	外部割込みコントローラ
MPU	0xE000ED90 〜 0xE000EDEF	メモリ保護ユニット
デバッグ	0xE000EDF0 〜 0xE000EEFF	デバッグ制御およびコンフィギュレーション

B2.2.1 システム制御ブロック (SCB)

ARMv7-M の主要な制御およびステータス機能は、SCS 内のシステム制御ブロックによって集中管理されます。SCB は、次の機能をサポートします。

- 各レベルのソフトウェアリセット制御
- 例外モデル用のベースアドレス管理（テーブルポインタの制御）
- システム例外の管理（NVIC によって処理される外部割込みを除く）
- コプロセッサへのアクセスのサポートを含む、各種の制御およびステータス機能
- 電力管理 - スリープのサポート
- フォルトステータス情報。フォルト処理の概要については P.B1-18「*フォルトの動作*」を参照して下さい。
- デバッグステータス情報。この情報は、デバッグ固有のレジスタ領域での制御およびステータスによって補完されます。デバッグの詳細については、第 C1 章 *「ARMv7-M デバッグ」*を参照して下さい。

レジスタの詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

B2.3 システムタイマ -SysTick

ARMv7-M には、アーキテクチャに組み込まれたシステムタイマである SysTick が含まれています。

SysTick は単純な 24 ビットの、書き込みクリア、デクリメント、ゼロでラップアラウンドするカウンタで、柔軟な制御機構を備えています。このカウンタは、次に示すようなさまざまな用途に使用できます。

- プログラム可能な周期（例えば、100Hz）で動作して SysTick ルーチンを起動する、RTOS チックタイマ。
- コアクロックを使用した高速アラームタイマ。
- 可変レートのアラームまたは信号タイマ。周期は、使用される参照クロックとカウンタのダイナミックレンジに依存します。
- 単純なカウンタ。ソフトウェアでこのカウンタを使用し、完了までの時間や消費時間を測定できます。
- 不一致消失した / 一致する期間に基づいた、内部クロックソースの制御。制御およびステータスレジスタの COUNTFLAG ビットフィールドを使用して、動的なクロック管理制御ループの一部として、設定された時間内にアクションが完了したかどうかを判断できます。

B2.3.1 動作の理論

タイマは、次の 4 つのレジスタで構成されます。

- 制御およびステータスカウンタは、クロックの構成、カウンタの許可、SysTick 割り込みの許可、カウンタのステータスの判断を行います。
- カウンタのリロード値は、カウンタのラップアラウンド値として使用されます。
- カウンタの現在の値
- 較正値レジスタ。10ms (100Hz) システムクロックに必要なプリロード値を示します。

タイマが許可されると、リロードされた値から 0 へカウントダウンを行います。0 に到達すると、次のクロックエッジで SysTick リロード値レジスタの値がリロード（ラップアラウンド）され、次のクロックからデクリメントが行われます。リロード値レジスタに 0 を書き込むと、次のラップアラウンド時にカウンタが禁止されます。カウンタが 0 に到達すると、COUNTFLAG ステータスビットがセットされます。COUNTFLAG ビットは、読み出し時にクリアされます。

現在値レジスタに書き込みを行うと、レジスタと COUNTFLAG ステータスビットがクリアされます。この書き込みで SysTick 例外ロジックがトリガされることはありません。読み出し時には、現在の値はレジスタがアクセスされたときの値です。

コアがデバッグ状態（停止中）の場合、カウントはデクリメントされません。タイマは参照クロックで動作します。参照クロックは、コアクロックまたは外部クロックソースです。

B2.3.2 SCS でのシステムタイマ レジスタのサポート

レジスタの詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

B2.4 ネスト型ベクタ割り込みコントローラ (NVIC)

ARMv7-M では、割り込みコントローラが例外モデルの一部として統合されています。割り込みコントローラの動作は、ARM の汎用割り込みコントローラ (GIC) 仕様に準拠しており、他のアーキテクチャバリアントおよび ARMv7 プロファイルで使用できるように拡張されています。

ARMv7-M の NVIC アーキテクチャは、最大 496 (IRQ[495:0]) の別々の割り込みをサポートしています。サポートされている外部割込みラインの数は、読み出し専用の割り込みコントローラタイプ レジスタ (ICTR) によって判定できます。このレジスタには、システム制御空間のアドレス 0xE000E004 でアクセスできます。NVIC 関連の汎用レジスタはすべて、P.B2-6「*システム制御空間 (SCS)*」に示されているように、システム制御空間のメモリブロックからアクセス可能です。

B2.4.1 動作の理論

ARMv7-M は、レベル感知およびパルス感知の割り込み動作をサポートしています。つまり、レベル感知およびパルス感知の割り込みの両方を処理可能です。パルス割り込みソースは、ラッチされて保留中になることを保証するため、コアクロックによって確実にサンプリングされるのに十分な時間だけ保持される必要があります。後続のパルスにより、割り込みがアクティブな間に再保留とすることができます。ただし、アクティブな期間に複数のパルスが発生すると、割り込みスケジューリングでは単一のイベントとしてのみ登録されます。

要約すると、次のようになります。

- 一クロック期間保持されるパルスは、エッジ感知割り込みと同様に動作します。これらは、割り込みがアクティブなときに再保留できます。

  注

  パルスは、AIRCR.VECTCLRACTIVE のアサート、または関連する例外からの復帰の前にクリアされる必要があります。これを行わない場合、割り込み信号はレベル感知入力として動作し、保留ビットが再度アサートされます。

- レベルベースの割り込みは、割り込みを保留にし、アクティブにします。割り込み処理ルーチン (ISR) は、その後でペリフェラルにアクセスし、レベルのアサートを解除できます。割り込みからの復帰時に割り込みが依然としてアサートされている場合、その割り込みは再度保留になります。

すべての NVIC 割り込みには、ARMv7-M 例外モデルとその優先度付けポリシーの一部として、プログラム可能な優先度の値と、対応する割り込み番号が設定されています。

NVIC は、次の機能をサポートしています。

- NVIC 割り込みは、対応する割り込みセットイネーブル、または割り込みクリアイネーブル レジスタのビットフィールドに書き込むことで、許可および禁止できます。このレジスタは、1 を書き込んで許可と 1 を書き込んでクリアのポリシーを使用しており、どちらのレジスタからも、対応する (32) 割り込みの現在の許可状態が読み出されます。

  割り込みが禁止されているとき、割り込みをアサートするとその割り込みは保留中になりますが、アクティブにはなりません。割り込みがアクティブであるときに禁止された場合、リセットまたは例外からの復帰によりクリアされるまで、その割り込みはアクティブ状態に保持されます。許可ビットをクリアすると、対応する割り込みは新たにアクティブになることが禁止されます。

  割り込み許可ビットは、対応する割り込みラインが存在しない場合は 0 に固定でき、対応する割り込みラインを禁止できない場合は 1 に固定できます。

- NVIC 割り込みは、割り込みを許可 / 禁止するために使用されるレジスタの補助ペアを使用して保留 / 保留解除できます。これらのレジスタを、それぞれセット保留レジスタ、クリア保留レジスタと呼びます。これらのレジスタは、1 を書き込んで許可、および 1 を書き込んでクリアのポリシーを使用しており、どちらのレジスタからも、対応する (32) 割り込みの現在の保留状態が読み出されます。クリア保留レジスタは、アクティブな割り込みの実行ステータスには影響しません。

  サポートされている各割り込みラインについて、ソフトウェア制御で割り込みに対応する保留ビットのセット、クリア、または両方をサポートするかどうかは IMPLEMENTATION DEFINED です。
- アクティブビットのステータスは、割り込みが非アクティブ、アクティブ、保留、またはアクティブかつ保留のどの状態であるかをソフトウェアで判定するために用意されています。
- NVIC 割り込みは、32 ビットレジスタ内の 8 ビットフィールドを更新することで優先度付けされます（各レジスタが 4 つの割り込みをサポートします）。優先度は、ARMv7-M の優先度付け方式に従って保守されます。

外部ハードウェアイベント、またはセット保留レジスタの対応するビットをセットする他に、ソフトウェアからソフトウェアトリガ割り込みレジスタに割り込み番号（ExceptionNumber - 16）を書き込んで、外部割込みを保留にすることもできます。

外部割込み入力の動作

割り込み入力信号に対応するエッジを決定するため、信号タイプが定義されています。信号のサンプルポイントを定義するために使用されるサブスクリプトを次に示します。

```
// DEFINITIONS

NVIC[] is an array of active high external interrupt input signals;
    // the type of signal (level or pulse) and its assertion level/sense is IMPLEMENTATION DEFINED
    // and might not be the same for all inputs

boolean Edge(integer INTNUM);    // Returns true if on a clock edge NVIC[INTNUM]
                                 // has changed from '0' to '1'
boolean NVIC_Pending[INTNUM];    // an array of pending status bits for the external interrupts
integer INTNUM;                  // the external interrupt number


    // The WriteToRegField helper function returns TRUE on a write of '1' event
    // to the field FieldNumber of the RegName register.

boolean WriteToRegField(register RegName, integer FieldNumber)


boolean ExceptionIN(integer INTNUM);   // returns TRUE if exception entry in progress
                                       // to activate INTNUM
boolean ExceptionOUT(integer INTNUM);  // returns TRUE if exception return in progress
                                       // from active INTNUM
```



```
// INTERRUPT INTERFACE

sampleInterruptHi = WriteToRegField(AIRCR, VECTCLRACTIVE) || ExceptionOUT(INTNUM);
sampleInterruptLo = WriteToRegField(ICPR, INTNUM);

InterruptAssertion   =  Edge(INTNUM) || (NVIC[INTNUM] && sampleInterruptHi);
InterruptDeassertion =  !NVIC[INTNUM] && sampleInterruptLo;



// NVIC BEHAVIOR

clearPend = ExceptionIN(INTNUM) || InterruptDeassertion;
setPend   = InterruptAssertion || WriteToRegField(ISPR, INTNUM);

if clearPend && setPend then
    IMPLEMENTATION DEFINED whether NVIC_Pending[INTNUM] is TRUE or FALSE;
else
    NVIC_Pending[INTNUM] = setPend || (NVIC_Pending[INTNUM] && !clearPend);
```

ARMv7-M での外部割込みには、次のような動作が推奨されています。

```
if ExceptionEntry(INTNUM) then
    PEND[INTNUM] = 0;    // clears the PENDING bit associated with INTNUM
elsif Edge(INTNUM, EdgeType) then
    PEND[INTNUM] = 1;    // sets the PENDING bit associated with INTNUM
elsif (AIRCR.VECTCLRACTIVE OR ExceptionReturn(INTNUM) ) AND NVIC[INTNUM] == '1' then
    PEND[INTNUM] = 1;    // sets the PENDING bit associated with INTNUM
elsif (AIRCR.VECTCLRACTIVE OR ICPR.INTNUM) AND NVIC[INTNUM] == '0') then
    PEND[INTNUM] = 0;    // clears the PENDING bit associated with INTNUM
```

B2.4.2 SCS での NVIC レジスタのサポート

レジスタの詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

B2.5 保護メモリシステム アーキテクチャ

ユーザ（非特権）およびスーパバイザ（特権）ソフトウェアモデルをサポートするためには、アクセス権限を制御するためのメモリ保護方式が必要です。ARMv7-M は、保護メモリシステムアーキテクチャ (PMSAv7) をサポートしています。PMSAv7 準拠のシステムのシステムアドレス空間は、メモリ保護ユニット (MPU) によって保護されます。保護されるメモリは一組の領域に分割され、サポートされる領域の数は実装定義です。PMSAv7 のサポートする領域の最小サイズは 32 バイトですが、4GB のアドレス空間に対して使用できるレジスタリソースは有限なため、この方式は本質的に粒度の粗い保護方式です。この保護方式は 100% 予測可能で、すべての制御情報はコアに密結合されているレジスタに保持されます。メモリアクセスは、MPU レジスタインタフェースのソフトウェア制御にのみ必要となります。

ARM7v-M での MPU サポートはオプションで、次に示すように、P.B2-2「*システム アドレスマップ*」で説明されているシステム メモリマップと共存します。

- MPU サポートによって、物理アドレスへのアクセス権制御が可能になります。MPU ではアドレス変換は行われません。
- MPU が禁止されている、または存在しない場合、システムは P.B2-3 表 B2-1 にリスト表記されているデフォルトのシステムメモリマップを採用します。MPU が許可されている場合、許可されている領域は次の条件で、システムアドレスマップの定義に使用されます。
  - 専用ペリフェラルバス (PPB) へのアクセスは、常にデフォルトのシステム アドレスマップを使用します。
  - ベクタアドレス テーブルからの例外ベクタ読み出しは、常にデフォルトのシステムアドレスマップを使用します。
  - MPU がシステム空間（アドレス 0xE0000000 以上）に対応するデフォルトのメモリマップ属性を変更する方法は制限されます。
    システム空間は、常に XN（実行不可）にマークされています。
    デフォルトがデバイスになっているシステム空間は、ストロングリオーダに変更できますが、ノーマルメモリにマップすることはできません。
  - 優先度 < 0 で実行されている例外（NMI、HardFault、および FAULTMASK がセットされている例外ハンドラ）は、MPU が許可または禁止されている状態で実行されるように構成可能です。
  - デフォルトのシステム メモリマップは、特権アクセス用のバックグラウンド領域を提供するように構成できます。
  - 複数の領域に一致するアドレスへのアクセスでは、一致するうちで最も大きい領域番号がアクセス属性として使用されます。
  - 領域のアドレス一致（MPU が許可されている場合）、またはバックグラウンド / デフォルト メモリマップ一致についてすべてのアクセス条件に一致しないアクセスは、フォルトを発生します。

B2.5.1 PMSAv7 準拠の MPU の動作

ARMv7-M は、MPU 領域サポートに関して、統合メモリモデルのみをサポートしています。許可されているすべての領域は、命令およびデータのアクセスをサポートしています。
領域のベースアドレス、サイズ、属性はすべて構成可能ですが、すべての領域がその領域のサイズでアラインしているという一般的な規則が適用されます。この規則は、次の式で表されます。

`RegionBaseAddress[(N-1) :0] = 0, where N is` $\log_2$`(SizeofRegion_in_bytes)`

メモリ領域は、2 のべき乗の範囲でサイズを変更できます。サポートされているサイズは 2^N で、5 ≤ N ≤ です。2 つの領域がオーバラップしている場合、領域番号が最も大きいレジスタが優先されます。

サブ領域のサポート

256 バイト以上の領域は、サイズ $2^{(N-3)}$ の最大 8 つのサブ領域に分割できます。領域内のサブ領域は、対応する領域属性レジスタによって個別に禁止できます（8 つの禁止ビットが存在します）。サブ領域が禁止されている場合、他の領域からのアクセス一致、またはバックグラウンドが許可されている場合は、バックグラウンド一致が必要です。アクセス一致が発生しない場合、フォルトが発生します。256 バイトより小さい領域には、サブ領域を設定できません。256 バイトよりも小さい領域については、サブ領域禁止フィールドは SBZ/UNP です。

ARMv7-M 固有のサポート

ARMv7-M は、標準の PMSAv7 メモリモデルに加え、次の拡張機能をサポートしています。

- 最適化された 2 レジスタの更新モデルにより、更新される領域を、MPU 領域ベースアドレス レジスタへの書き込みによって選択できます。この最適化は、最初の 16 のメモリ領域（0 ≤ 領域番号 ≤ 0xF）にのみ適用されます。
- MPU 領域ベースアドレス レジスタと MPU 領域属性およびサイズレジスタのペアは、3 つの連続したデュアルワードの位置にエイリアスされています。2 レジスタの更新モデルを使用すると、単一の STM 複数ワードストア命令を使用して適切な偶数番号のワードに書き込むことで、最大 4 つの領域を変更できます。

B2.5.2 SCS での PMSAv7 レジスタサポート

レジスタの詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

第 B3 章
ARMv7-M システム命令

これまでに説明したように、ARMv7-M は Thumb 状態でのみ命令を実行します。サポートされている命令の完全なリストについては、P.A6-17「*ARMv7-M Thumb 命令のアルファベット順リスト*」を参照して下さい。ソフトウェア制御のもとで専用レジスタの読み出しと書き込みをサポートするため、ARMv7-M には次の 3 つのシステム命令が用意されています。

CPS

MRS

MSR

B3.1 ARMv7-M システム命令のアルファベット順リスト

このセクションでは、次の ARMv7-M システム命令の定義について説明します。

- *CPS*
- *MRS*：P.B3-4
- *MSR（レジスタ）*：P.B3-8[1]

B3.1.1 CPS

CPS（プロセッサ状態変更）は、1 つまたは複数の専用レジスタ PRIMASK および FAULTMASK の値を変更します。

エンコード T1 ARMv6T2, ARMv7.

CPS<effect> <iflags>

IT ブロックでは許可されない

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	(0)	I	F

```
enable = (im == '0');  disable = (im == '1');
affectPRI = (I == '1');  affectFAULT = (F == '1');
if InITBlock() then UNPREDICTABLE;
```

アセンブラ構文

CPS<effect><q> <iflags>

where:

<effect> PRIMASK および FAULTMASK で必要な効果を指定します。次のいずれかを指定します。

IE 割り込み許可。指定のビットが 0 にセットされます。

ID 割り込み禁止。指定のビットが 1 にセットされます。

<q> 詳細については、P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。CPS は常に無条件命令です。

<iflags> 次のうち 1 つ以上のシーケンスで、影響を受けるマスクを指定します。

i PRIMASK。1 にセットすると、現在の優先度を 0 に引き上げます。このレジスタは 1 ビットで、特権アクセスのみをサポートします。

f FAULTMASK。1 にセットすると、現在の優先度を -1（HardFault と同じ）に引き上げます。このレジスタは 1 ビットで、優先度が -1 より低い特権コードによってのみセットできます。このレジスタは、NMI 以外の例外から復帰するときに自動的にクリアされます。

1. MSR（イミディエート）は、他の ARMv7 プロファイルや以前のアーキテクチャバリアントでは有効な命令です。MSR（イミディエート）エンコードは ARMv7-M では未定義です。

動作

```
EncodingSpecificOperations();
if CurrentModeIsPrivileged() then
    if enable then
        if affectPRI then PRIMASK<0> = ‘0’;
        if affectFAULT then FAULTMASK<0> = ‘0’;
    if disable then
        if affectPRI then PRIMASK<0> = ‘1’;
        if affectFAULT && ExecutionPriority > -1 then FAULTMASK<0> = ‘1’;
```

例外

なし

注

特権　　ユーザモードのコードからこれらのマスクに書き込みを試みた場合、すべて無視されます。

マスクと CPS

CPSIE および CPSID 命令は、MSR 命令の使用と等価です。

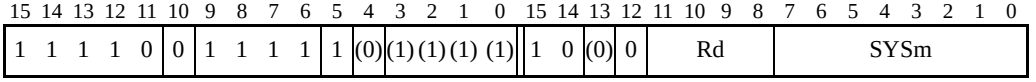
- CPSIE i 命令は、PRIMASK へ 1 を書き込むのと等価です。
- CPSID i 命令は、PRIMASK へ 1 を書き込むのと等価です。
- CPSIE f 命令は、FAULTMASK へ 0 を書き込むのと等価です。
- CPSID f 命令は、FAULTMASK へ 1 を書き込むのと等価です。

B3.1.2 MRS

MRS（特殊レジスタからレジスタへの移動）は、選択された専用レジスタの値を汎用レジスタに移動します。

エンコード T1 ARMv6T2, ARMv7.

MRS<c> <Rd>,<spec_reg>




```
d = UInt(Rd);
if BadReg(d) || (SYSm == unused) then UNPREDICTABLE;   // see SYSm encoding under Assembler Syntax
```

アセンブラ構文

MRS<c><q> <Rd>, <spec_reg>

where:

<c><q> 詳細については、P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rd> デスティネーションレジスタを指定します。

<spec_reg> SYSm にエンコードされ、次のいずれかを指定します。

特殊レジスタ	内容	SYSm の値
APSR	前の命令からのフラグ	0
IAPSR	IPSR と APSR の組み合わせ	1
EAPSR	EPSR と APSR の組み合わせ	2
XPSR	3 つの PSR レジスタすべての組み合わせ	3
IPSR	割り込みステータスレジスタ	5
EPSR	実行ステータスレジスタ	6
IEPSR	IPSR と EPSR の組み合わせ	7
MSP	メイン スタックポインタ	8
PSP	プロセス スタックポインタ	9
PRIMASK	構成可能な例外をマスクするレジスタ	16 [a]
BASEPRI	ベース優先度レジスタ	17 [b]

特殊レジスタ	内容	SYSm の値
BASEPRI_MAX	読み出し時の BASEPRI のエイリアスとして動作	18 [c]
FAULTMASK	優先度を HardFault レベルへ引き上げるためのレジスタ	19 [d]
CONTROL	専用制御レジスタ	20 [e]
RSVD	予約	unused

a. 1 にセットすると、現在の優先度を 0 に引き上げます。このレジスタは 1 ビットです。
b. 現在の横取り優先度マスクを、0 ～ N の値に変更します。0 は、マスクが禁止されていることを意味します。このレジスタは、値 (1 ～ N) が実行中の命令ストリームのマスクされていない優先度レベルよりも低い（優先度が高い）場合にのみ効果があります。
このレジスタは 8 つまでのビットを持つことができ（サポートされている優先度の数によって変化）、他の優先度レジスタと正確に同じフォーマットです。
このレジスタは、PRIGROUP（2 進小数点）フィールドの影響を受けます。詳細については、P.B1-14「*例外の優先度と横取り*」を参照して下さい。優先度の横取り部分のみが、BASEPRI によってマスクに使用されます。
c. MSR 命令とともに使用された場合、条件付き書き込みを実行します。
d. 1 にセットすると、現在の優先度を -1（HardFault と同じ）に引き上げます。このレジスタは優先度が -1 より低い特権コード（NMI または HardFault 以外）のみがセットでき、NMI 以外の例外からの復帰時には自動的にクリアされます。このレジスタは 1 ビットです。
e. 制御レジスタは、次のビットで構成されています。
[0] = スレッドモードの権限。0 は特権モード、1 は非特権（ユーザ）モードを意味します。リセット時の値は 0 です。
[1] = 現在のスタックポインタ。0 はメインスタック (MSP) 、1 は代替スタック（スレッドモードでは PSP、ハンドラモードでは予約）を意味します。リセット時の値は 0 です。

動作

```
if ConditionPassed() then
    R[d] = 0;
    case  SYSm<7:3> of
        when ‘00000’
            if SYSm<0> == ‘1’ and CurrentModeIsPrivileged() then
                R[d]<8:0> = IPSR<8:0>;
            if SYSm<1> == ‘1’ then
                R[d]<26:24> = ‘000’;     /* EPSR reads as zero */
                R[d]<15:10> = ‘000000’;
            if SYSm<2> == ‘0’ then
                R[d]<31:27> = APSR<31:27>;
        when ‘00001’
            if CurrentModeIsPrivileged() then
                case SYSm<2:0> of
                    when ‘000’
                        R[d] = MSP;
                    when ‘001’
                        R[d] = PSP;
        when ‘00010’
            case SYSm<2:0> of
                when ‘000’
                    R[d]<0> = if CurrentModeIsPrivileged() then
                        PRIMASK<0> else ‘0’;
                when ‘001’
                    R[d]<7:0> = if CurrentModeIsPrivileged() then
                        BASEPRI<7:0> else ‘00000000’;
                when ‘010’
                    R[d]<7:0> = if CurrentModeIsPrivileged() then
                        BASEPRI<7:0> else ‘00000000’;
                when ‘011’
                    R[d]<0> = if CurrentModeIsPrivileged() then
                        FAULTMASK<0> else ‘0’;
                when ‘100’
                    R[d]<1:0> = CONTROL<1:0>;
```

例外

なし

注

特権　ユーザコードでいずれかのスタックポインタ、または IPSR の読み出しを試みると、0 が返されます。

EPSR　EPSR のビットはどれも、通常の実行時は読み出し不可です。MRS を使用して読み出すと、すべて 0 として読み出されます（ホールトデバッグでは、レジスタ転送機構により読み出すことができます）。

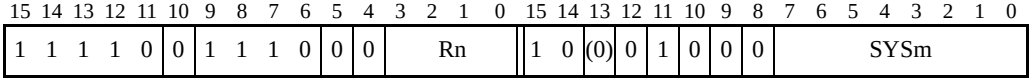
ビット位置　PSR のビット位置は、P.B1-7「*専用プログラム ステータスレジスタ (xPSR)*」で定義されています。

B3.1.3 MSR（レジスタ）

MSR（ARM レジスタから特殊レジスタへの移動）は、汎用レジスタの値を選択された専用レジスタへ移動します。

エンコード T1 ARMv6T2, ARMv7.

MSR<c> <spec_reg>,<Rn>




```
n = UInt(Rn);
if BadReg(n) || (SYSm == unused) then UNPREDICTABLE;   // see SYSm value encoding below
```

アセンブラ構文

```
MSR<c><q>  <spec_reg>, <Rn>
```

where:

<c><q> 詳細については、P.A6-7「*標準アセンブラ構文のフィールド*」を参照して下さい。

<Rn> 特殊レジスタの内容を受け取る汎用レジスタ

<spec_reg> SYSm にエンコードされ、次のいずれかを指定します。

特殊レジスタ	内容	SYSm の値
APSR	前の命令からのフラグ	0
IAPSR	IPSR と APSR の組み合わせ	1
EAPSR	EPSR と APSR の組み合わせ	2
XPSR	3 つの PSR レジスタすべての組み合わせ	3
IPSR	割り込みステータスレジスタ	5
EPSR	実行ステータスレジスタ（0 として読み出されます。注を参照）	6
IEPSR	IPSR と EPSR の組み合わせ	7
MSP	メイン スタックポインタ	8
PSP	プロセス スタックポインタ	9
PRIMASK	構成可能な例外をマスクするレジスタ	16 [a]
BASEPRI	ベース優先度レジスタ	17 [b]

特殊レジスタ	内容	SYSm の値
BASEPRI_MAX	書き込み時に BASEPRI を引き上げますが、引き下げは行わない	18 [c]
FAULTMASK	優先度を HardFault レベルへ引き上げるためのレジスタ	19 [d]
CONTROL	専用制御レジスタ	20 [e]
RSVD	予約	unused

a. 1 にセットすると、現在の優先度を 0 に引き上げます。このレジスタは 1 ビットです。
b. 現在の横取り優先度マスクを、0 ～ N の値に変更します。0 は、マスクが禁止されていることを意味します。このレジスタは、値 (1 ～ N) が実行中の命令ストリームのマスクされていない優先度レベルよりも低い（優先度が高い）場合にのみ効果があります。
このレジスタは 8 つまでのビットを持つことができ（サポートされている優先度の数によって変化）、他の優先度レジスタと正確に同じフォーマットです。
このレジスタは、PRIGROUP（2 進小数点）フィールドの影響を受けます。詳細については、P.B1-14「*例外の優先度と横取り*」を参照して下さい。優先度の横取り部分のみが、BASEPRI によってマスクに使用されます。
c. MSR 命令とともに使用された場合、条件付き書き込みを実行します。BASEPRI の値は、新しい優先度が現在の BASEPRI の値より高い（数値が小さい）場合にのみ更新されます。
BASEPRI では、0 は禁止を意味する特別な値です。BASEPRI が 0 の場合、新しい値は常に受け付けられます。新しい値が 0 の場合、決して受け付けられません。つまり、
BASEPRI_MAX は必ず BASEPRI を許可しますが、禁止することはできません。PRIGROUP は、比較または書き込まれる値に影響しません。レジスタのすべてのビットは、比較されて条件付きで書き込まれます。
d. このレジスタは、1 にセットすると、現在の優先度を -1（HardFault と同じ）に引き上げます。このレジスタは優先度が -1 より低い特権コード（NMI または HardFault 以外）のみがセットでき、NMI 以外の例外からの復帰時には自動的にクリアされます。このレジスタは 1 ビットです。CPS 命令を使用して FAULTMASK レジスタを更新することもできます。
e. 制御レジスタは、次のビットで構成されています。
[0] = スレッドモードの権限。0 は特権モード、1 は非特権（ユーザ）モードを意味します。リセット時の値は 0 です。
[1] = 現在のスタックポインタ。0 はメインスタック (MSP) 、1 は代替スタック（スレッドモードでは PSP、ハンドラモードでは RESERVED）を意味します。リセット時の値は 0 です。

動作

```
if ConditionPassed() then
    case SYSm<7:3> of
        when ‘00000’
            if SYSm<2> == ‘0’ then
                APSR<31:27> = R[n]<31:27>;
        when ‘00001’
            if CurrentModeIsPrivileged() then
                case SYSm<2:0> of
                    when ‘000’
                        MSP = R[n];
                    when ‘001’
                        PSP = R[n];
        when ‘00010’
            case SYSm<2:0> of
                when ‘000’
                    if CurrentModeIsPrivileged() then PRIMASK<0> = R[n]<0>;
                when ‘001’
                    if CurrentModeIsPrivileged() then BASEPRI<7:0> = R[n]<7:0>;
                when ‘010’
                    if CurrentModeIsPrivileged() &&
                    (R[n]<7:0> != ‘00000000’) &&
                    (R[n]<7:0> < BASEPRI<7:0> || BASEPRI<7:0> == ‘00000000’) then
                        BASEPRI<7:0> = R[n]<7:0>;
                when ‘011’
                    if CurrentModeIsPrivileged() &&
                    (ExecutionPriority > -1) then
                        FAULTMASK<0> = R[n]<0>;
                when ‘100’
                    if CurrentModeIsPrivileged() then
                        CONTROL<0> = R[n]<0>;
                        If Mode == Thread then CONTROL<1> = R[n]<1>;
```

例外

なし

注

特権　非特権スレッドモードからのスタックポインタ、EPSR、IPSR、マスク、CONTROL への書き込みは無視されます。特権スレッドモードのソフトウェアが CONTROL[0] に 0 を書き込んだ場合、コアは非特権スレッドモード（ユーザ）での実行に切り替わり、それ以後は専用レジスタへの書き込みが禁止されます。

スレッドモードで特権 => 非特権の遷移を行った後では、命令フェッチが正確であることを保証するために ISB 命令が必要です。

IPSR　現在定義されている IPSR フィールドは書き込み不可です。特権コードでこれらのフィールドに書き込みを行った場合、無視されます（何も影響しません）。

EPSR　現在定義されている EPSR フィールドは書き込み不可です。特権コードでこれらのフィールドに書き込みを行った場合、無視されます（何も影響しません）。

ビット位置　PSR ビットは、より大きい xPSR 組み合わせ内での位置に従って、各 PSR に配置されます。これらの位置の定義については、P.B1-7「*専用プログラム ステータス レジスタ (xPSR)*」を参照して下さい。

パート C

デバッグアーキテクチャ

第 C1 章
ARMv7-M デバッグ

本章では、ARMv7-M でサポートされているデバッグ機能の概要を紹介します。本章は、デバイスのテクニカル リファレンスマニュアル (TRM) と同時に読むことを意図して作成されています。TRM には、レジスタインタフェースやプログラミングモデルなど、デバイスのデバッグ機構の詳細が記載されています。本章は以下のセクションから構成されています。

本章はプロファイル固有です。ARMv7-M には、ARMv7 アーキテクチャ内でこのプロファイルに固有の、いくつかのデバッグ機能が含まれています。

C1.1 デバッグの概要

デバッグサポートは、ARM アーキテクチャの主要な要素です。ARMv7-M では、侵襲性および非侵襲性の両方の手法を含む、広範なデバッグ手法が提供されています。

侵襲性のデバッグ技法には次のものがあります。

- コアの停止やブレークポイントの実行などを行う機能（実行 - 停止モデル）
- DebugMonitor 例外を使用するデバッグコード（コアの停止より侵襲性が少ない）

非侵襲性のデバッグ技法には次のものがあります。

- 計装トレースマクロセル (ITM) への書き込みによるアプリケーショントレース（ごく低いレベルの侵襲）
- ハードウェアサポートによる非侵襲性のトレースとプロファイリング

デバッグは通常、DAP 経由でアクセスされます（P.C1-4「*デバッグアクセス ポート (DAP)*」参照）。これによって、プロセッサが実行中、停止中、リセット保持中のいずれの状態でも、デバッグおよびシステムメモリへのアクセスが可能になります。コアが停止しているとき、コアはデバッグ状態です。

ソフトウェアベース、および非侵襲性のハードウェアデバッグ機能として、次のものがサポートされています。

- 計装トレースマクロセル (ITM) を使用した、高レベルのトレースおよびログ出力。この機能は、一定の低侵襲性オーバヘッド（レジスタへのノンブロッキング書き込み）を使用し、RTOS、アプリケーション、例外ハンドラ /ISR に追加可能です。必要なところではプローブの影響を避けながら、製品コード中にその命令を残しておくことができます。
- 関連するタイミング情報など、各種のシステムイベントのプロファイリング。これには、割り込みやスリープ機能に関連するコアクロックのカウント数監視が含まれます。
- ロード / ストア、命令フォルディング、CPI 統計に関連する PC のサンプリングとイベントのカウント
- データトレース

システム制御空間 (SCS) 内のデバッグ制御ブロック (DCB) の場合と同様に、デバッグに関連する他のリソースは、ARMv7-M システム アドレスマップの専用ペリフェラルバス (PPB) 領域内にある、決まった 4KB のアドレス領域に割り当てられます。

- ソフトウェアのプロファイリング用の計装トレースマクロセル (ITM)
- データウォッチポイントおよびトレース (DWT)。ウォッチポイントのサポート、パフォーマンス監視用のプログラムカウンタ サンプリング、エンベデッドトレーストリガの制御を行います。
- フラッシュパッチおよびブレークポイント (FPB) ブロック。このブロックは、ROM（フラッシュメモリ）のセクションを RAM の領域にリマップし、ROM のコードにブレークポイントを設定できます。この機能は、デバッグに使用する他、製品の ROM を現場で更新または修正する必要がある場合、コードやデータへのパッチを提供するために使用できます。
- エンベデッドトレースマクロセル (ETM)。このブロックはオプションで、命令のトレースに使用されます。
- トレースポート インタフェースユニット (TPIU) 。このブロックはオプションで、ITM、DWT、ETM（該当する場合）トレース機能へのピンインタフェースを提供します。

- ROM テーブル。エントリのテーブルで、実装によってサポートされているデバッグインフラストラクチャを特定する機構を提供します。

ITM、DWT、FPB、DCB、およびトレースサポートのアドレス範囲のリストを表 C1-1 に示します。

注

最小システムには、ここに記載されているデバッグ機能がすべて含まれていない可能性があります。詳細については、P.C1-9「*ARMv7-M でのデバッグサポート*」を参照して下さい。

表 C1-1 PPB デバッグに関連する領域

専用ペリフェラルバス（アドレス範囲 0xE0000000 ～ 0xE00FFFFF）		
グループ	アドレスのオフセット範囲	注
計装トレースマクロセル (ITM)	0xE0000000-0xE0000FFF	プロファイリングとパフォーマンス監視のサポート
データウォッチポイントおよびトレース (DWT)	0xE0001000-0xE0001FFF	トレースサポート用の制御を含む
フラッシュパッチとブレークポイントユニット (FPB)	0xE0002000-0xE0002FFF	オプションのブロック
SCS：システム制御ブロック (SCB)	0xE000ED00-0xE000ED8F	SCB：汎用の制御機能
SCS：デバッグ制御ブロック (DCB)	0xE000EDF0-0xE000EEFF	デバッグ制御および構成
トレースポート インタフェースユニット (TPIU)	0xE0040000-0xE0040FFF	オプションのトレースやシリアルワイヤ ビューアのサポート（注記参照）
エンベデッドトレースマクロセル (ETM)	0xE0041000-0xE0041FFF	オプションの命令トレース機能
ARMv7-M の ROM テーブル	0xE00FF000-0xE00FFFFF	自動構成用に DAP からアクセス可能

表 C1-1 に関する注記

- ITM スティミュラスポートへの書き込みでは、ITM 機能が禁止されているか存在しない場合に例外を発生せず、この機能がアプリケーションコードから透過的であることを保証する必要があります。
- SCB については、P.B2-7「*システム制御ブロック (SCB)*」を参照して下さい。
- データトレース、アプリケーショントレース、プロファイリングには、DWT と ITM の他に TPIU が必要です。
- TPIU は複雑なデバッグシステムの共有リソースとすることができますが、ITM スティミュラス、または ETM と DWT トレースイベント出力の可視性が必要でない場合は省略することもできます。TPIU を共有リソースとする場合、PPB メモリマップ内に置いてローカルプロセッサの制御下とするか、外部システムリソースとして別の場所から制御します。

C1.2 デバッグアクセス ポート (DAP)

デバッグアクセスは、デバッグアクセス ポート (DAP) 経由で行います。JTAG デバッグポート (JTAG-DP) またはシリアルワイヤ デバッグポート (SW-DP) を使用できます。DAP の仕様には、どのデバッグリソースが使用可能かをシステムに問い合わせる方法と、ARMv7-M デバイスへアクセスする方法の詳細が含まれています。有効な ARMv7-M システムのインスタンスには、表 C1-3 に示すような情報の ROM テーブルが含まれています。ROM テーブルエントリの一般的なフォーマットを表 C1-2 に示します。

デバッガは、DAP インタフェースを使用して、システムに対してメモリアクセス ポート (MEM-AP) に関する問い合わせを実行できます。メモリアクセス ポートの BASE レジスタは、ROM テーブル（または、ROM テーブル階層内の一連の ROM テーブル）のアドレスを示しています。アドレスを取得した後で、メモリアクセス ポートを使用して ROM テーブルのエントリをフェッチできます。

表 C1-2 ROM テーブルエントリのフォーマット

ビット	名前	説明
[31:12]	アドレスオフセット	ROM ベースアドレスからコンポーネントへの相対的なベースアドレスオフセット（符号付き）
[11:2.]	予約	UNK/SBZP
[1]	フォーマット	有効なテーブルエントリの場合、1 が読み出されます。
[0]	エントリの存在	1：有効なテーブルエントリ 0：(かつ、ビット [31:1] が 0 に等しくない場合)、テーブルエントリを無視します。[a]

a. ARMv7-M では、テーブル終了マーカの前に NULL エントリが必要な場合、NULL エントリとして 0x00000002 を使うことを推奨します。

ARMv7-M では、すべてのアドレスオフセットは負の値です。

表 C1-3 ARMv7-M の DAP でアクセス可能な ROM テーブル

オフセッ	値	名前	説明
0x000	0xFFF0F003	ROMSCS	0xE000E000 にある SCS を指し示します。
0x004	0xFFF02002 または 0xFFF02003	ROMDWT	0xE0001000 にあるデータウォッチポイントおよびトレースブロックを指し示します。 DWT が存在する場合、ビット [0] がセットされています。
0x008	0xFFF03002 または 0xFFF03003	ROMFPB	0xE0002000 にあるフラッシュパッチとブレークポイントブロックを指し示します。 FPB が存在する場合、ビット [0] がセットされています。
0x00C	0xFFF01002 または 0xFFF01003	ROMITM[a]	0xE0000000 にある計装トレースブロックを指し示します。 ITM が存在する場合、ビット [0] がセットされています。
0x010	0xFFF41002 または 0xFFF41003	ROMTPIU[b]	トレースポート インタフェースユニットを指し示します。 TPIU が存在し、プロセッサから専用ペリフェラルバス (PPB) でアクセス可能な場合、ビット [0] がセットされています。
0x014	0xFFF42002 または 0xFFF42003	ROMETM[b]	エンベデッドトレースマクロセル ブロックを指し示します。 ETM が存在し、プロセッサから PPB でアクセス可能な場合、ビット [0] がセットされています。

表 C1-3 ARMv7-M の DAP でアクセス可能な ROM テーブル

オフセッ	値	名前	説明
0x018	0	終了	テーブル終了マーカ。テーブルが、他のシステムデバッグ リソースへのポインタにより拡張されるかどうかは実装定義です。テーブルのエントリの最後は、常に NULL エントリです。
0x020 ～ 0xFC8		未使用	RAZ
0xFCC	0x00000001	SYSTEM ACCESS[c]	ビット [0] がセットされている場合、ROM テーブルのリストに含まれている以外のリソースが、同一の 32 ビットアドレス空間内で DAP を経由してアクセス可能なことを示します。
0xFD0	IMP DEF	PID4	CIDx の値は ROM テーブル用に完全に定義されており、CoreSight に準拠しています。 PIDx の値は、CoreSight 準拠または RAZ の必要があります。 CoreSight：ARM のシステムデバッグ アーキテクチャ
0xFD4	0	PID5	
0xFD8	0	PID6	
0xFDC	0	PID7	
0xFE0	IMP DEF	PID0	
0xFE4	IMP DEF	PID1	
0xFE8	IMP DEF	PID2	
0xFEC	IMP DEF	PID3	
0xFF0	0x0000000D	CID0	
0xFF4	0x00000010	CID1	
0xFF8	0x00000005	CID2	
0xFFC	0x000000B1	CID3	

a. アクセスにより、存在しないメモリの例外が発生することはありません。
b. 共有リソースがローカルプロセッサにより管理されるか、別のリソースにより管理されるかは実装定義です。
c. この位置は、以前は MEMTYPE と呼ばれていました。

DAP を使用して ARMv7-M のデバッグにアクセスし、有効にするためのイベントの基本的なシーケンスは次のとおりです。

- DAP デバッグポート制御レジスタ中のデバッグロジック用のパワーアップビットを有効にします。
- 適切な DAP メモリアクセス ポート制御レジスタが、ワードアクセス用に有効になっていることを確認します（ユニプロセッサシステムでは、この状態がデフォルトです）。
- ホールトデバッグが必要な場合、次の操作を行います。
  — デバッグホールト制御およびステータスレジスタ (DHCSR) で、C_DEBUGEN ビットをセットします。

ターゲットを直ちに停止する場合、次の操作を行います。

— 同じレジスタの C_HALT ビットをセットします。

— DHCSR の S_HALT ビットを読み返し、ターゲットがデバッグ状態で停止していることを確認します。

それ以外の場合、モニタデバッグが必要であれば次の操作を行います。

— DEMCR で DebugMonitor 例外を許可します。

注

DebugMonitor 例外が発生するには、C_DEBUGEN がクリアされている必要があります。C_DEBUGEN がセットされている場合、ホールトデバッグ動作が DebugMonitor 例外より優先されます。

- ウォッチポイントおよびトレース機能を使用する場合、デバッグ例外およびモニタ制御レジスタ (DEMCR) の TRCENA ビットをセットします。

警告

システム制御および構成フィールド（特に、SCB のレジスタ）は、ソフトウェアの実行中に DAP 経由で変更できます。例えば、動的な更新用に設計されたリソースは変更可能ですが、アプリケーションとデバッガの両方が同じ、または関連するリソースを更新すると、望ましくない副作用が発生することがあります。このような方法で、DAP により実行中のシステムを更新した結果についての保証はありません。システムの動作に関しては、予測不能よりも悪い結果となる可能性があります。

一般に、関連するコンテキストの問題を回避するため、MPU または FPB アドレスのリマッピング変更を、ソフトウェアの実行中にデバッガで行うことは避けて下さい。

C1.2.1 デバッグレジスタへのアクセスに適用される一般的な規則

専用ペリフェラルバス (PPB) 、アドレス範囲 0xE0000000 ～ 0xE0100000 は、次のような一般的な規則をサポートしています。

- この領域はストロングリオーダメモリとして定義されています。詳細については、P.A3-26「*ストロングリオーダメモリ*」および P.A3-27「*メモリアクセスの制約事項*」を参照して下さい。
- レジスタは、プロセッサのエンディアン形式に関係なく、常にリトルエンディアンとしてアクセスされます。
- デバッグレジスタに対しては、ワードアクセスのみが可能です。バイトおよびハーフワードアクセスの結果は予測不能です。
- セットするという用語は、値として 1 を割り当てることを意味します。クリアするという用語は、0 を割り当てることを意味します。これらの用語が複数のビットに対して使用される場合、すべてのビットに同じ値が割り当てられます。
- 禁止する という用語は、ビットに値として 0 を割り当てることを、許可するという用語は、ビットに 1 を割り当てることを意味します。
- 予約されているレジスタまたはビットフィールドの値は UNK/SBZP と見なされます。

PPB への非特権（ユーザ）アクセスは、特記されていない限り BusFault エラーを引き起こします。特記すべき例外には次のものがあります。

- 構成制御レジスタの制御ビットをプログラムすると、システム制御空間のソフトウェアトリガ割り込みレジスタへの非特権アクセスを許可できます。
- デバッグ関連のリソース（DWT、ITM、FPB、ETM、TPIU の各ブロック）については、特記されていない限りユーザアクセスの読み出しは未知で、書き込みは無視されます。

C1.3 ARMv7-M のデバッグ機能の概要

ARMv7-M には、このプロファイル用に特に設計されたデバッグモデルが定義されています。ARMv7-M のデバッグモデルでは、メモリマップ中に制御と構成が統合されています。デバッグ アクセス ポートは、ホストデバッガへのインタフェースを提供します。ARMv7-M 内のデバッグ リソースを P.C1-3 表 C1-1 に示します。

ARMv7-M では、次のようなデバッグ関連の機能がサポートされています。

- ローカルリセット。詳細については P.B1-12「*サポートされている例外の概要*」を参照して下さい。この機能はコアをリセットし、リセットイベントのデバッグをサポートします。
- コアの停止。制御レジスタは、コアの停止をサポートしています。これは、外部信号のアサート、BKPT 命令の実行、またはデバッグイベント（リセット時に発生するよう構成されている場合や、ISR の開始または終了）により、非同期に発生する可能性があります。
- 割り込みマスクあり、またはなしでのステップ実行
- 割り込みマスクあり、またはなしでの実行
- レジスタへのアクセス。DCB は、停止中のコアのレジスタに対する読み出しと書き込みを含むデバッグ要求をサポートしています。
- SCS リソースを経由しての、例外に関係する情報へのアクセス。例として、現在実行中の例外（存在する場合）、アクティブリスト、保留中リスト、および最も優先度の高い保留中の例外にアクセスできます。
- ソフトウェアブレークポイント。BKPT 命令がサポートされています。
- ハードウェアブレークポイント、ハードウェアウォッチポイント、コードメモリ位置のリマッピング
- DAP 経由でのすべてのメモリに対するアクセス
- プロファイリングのサポート。PC サンプリングがサポートされています。
- 命令トレースのサポートと、バス監視やクロストリガ機能など他のシステムデバッグ機能の追加のサポート。ETM 命令のトレースには、マルチワイヤのトレースポート インタフェースユニット (TPIU) が必要です。
- アプリケーションおよびデータのトレース。これらは、低ピン数のシリアルワイヤ ビューア (SWV) またはマルチワイヤ TPIU によってサポートできます。

注

ARM のシステムデバッグ アーキテクチャは CoreSight と呼ばれ、広範なデバッグ制御、キャプチャ、システムインタフェース ブロックが組み込まれています。ARMv7-M では CoreSight への準拠は必須ではありません。この仕様では、DWT、ITM、TPIU、FPB ブロックのレジスタ定義とアドレス空間の割り当ては互換性があります。ARMv7-M では、必要に応じてこれらのブロックに CoreSight ID と管理レジスタを拡張し、CoreSight トポロジの検出と操作のサポートを追加することができます。

C1.3.1 ARMv7-M でのデバッグサポート

ARMv7-M は、包括的なデバッグ機能のセットをサポートしています。次のビットフィールドを使用して、設計に含まれているデバッグサポートのレベルを判定できます。

- ROMDWT[0] が 0 の場合、DWT はサポートされていません。それ以外の場合、DEMCR.TRCENA == 1 で、
  - DWT_CTRL.NOTRCPKT == "1" の場合、DWT トレースサンプリングや例外トレースはサポートされていません。
  - DWT_CTRL.NOEXTTRIG == "1" の場合、**CMPMATCH[N]** はサポートされていません。
  - DWT_CTRL.NOCYCCNT == "1" の場合、サイクルカウンタはサポートされていません。
  - DWT_CTRL.NOPRFCNT == "1" の場合、プロファイリングカウンタはサポートされていません。
- ROMITM[0] が 0 の場合、ITM はサポートされていません。
- ROMFPB[0] が 0 の場合、FPB はサポートされていません。
- ROMETM[0] が 0 の場合、ETM はサポートされていません。
- DWT と ITM のどちらもサポートされていない場合、DEMCR.TRCENA は RAZ/WI です。
- FP_REMAP[29] が 0 の場合、FPB はブレークポイント機能のみをサポートしています。

注

DWT または FPB でサポートされているコンパレータの数は、それぞれ DWT_CTRL および FP_CTRL レジスタのビットフィールドで判定できます。

デバッグの推奨レベル

ARMv7-M のデバッグ機構には、3 つの推奨レベルがあります。

- DebugMonitor 例外のみをサポートする最小レベル
- DAP が必要でホールトデバッグをサポートする基本レベル
- 上記に加えて完全な機能の ITM、DWT、FPB サポートを含む総合レベル

ARMv7-M の最小レベルのデバッグでは、コアアクセス（DAP なし）と、次の方法による DebugMonitor 例外のみがサポートされます。

- BKPT 命令

  注

  DebugMonitor 例外が禁止されているとき、この例外は HardFault 例外に昇格されます。

- モニタステップ実行
- **EDBGRQ** からのモニタ開始

基本レベルのサポートとして、次の機能構成が定義されています。

- DAP およびホールトデバッグのサポート

- ITM サポートなし -ROMITM[0] == "0"、P.C1-4「*ARMv7-M の DAP でアクセス可能な ROM テーブル*」参照

  注

  ITM スティミュラスポートへの書き込みでは、ITM 機能が禁止されているか存在しない場合に例外を発生せず、この機能がアプリケーションコードから透過的であることを保証する必要があります。

- FPB 内の 2 つのブレークポイント（リマッピングのサポートなし）
- DWT 内の 1 つのウォッチポイント（トレースサンプリングや外部一致信号 **(CMPMATCH[N])** の生成なし）
- リスト表記されている FPB および DWT イベントとともに、最小レベルのデバッグ機能のデバッグモニタ サポート

総合的なレベルのサポートに準拠するには、次の機能が必要です。

- DebugMonitor 例外とホールトデバッグがサポートされている。
- ROMITM[0] != "0" の場合
  - 最低 8 つのスティミュラスポート レジスタ
- ROMDWT[0] != "0" の場合
  - 最低 1 つのウォッチポイントのサポート
  - DWT_CTRL.NOTRCPKT == "0"
  - DWT_CTRL.NOCYCCNT == "0"
  - DWT_CTRL.NOPRFCNT == "0"
  - DWT_CTRL.NOEXTTRIG は実装定義

    ROMETM[0] == "1" の場合、**CMPMATCH[N]** サポートが必要
- ROMFPB[0] != "0" の場合
  - 最低 2 つのブレークポイントのサポート
  - FP_REMAP[29] != "0"

C1.4 デバッグとリセット

ARMv7-M では、P.B1-12「*サポートされている例外の概要*」に示されているような 2 つのレベルのリセットが定義されています。リセットハンドラの実行が開始されると、コアの停止（デバッグ状態に入る）が可能になります。デバッグ例外およびモニタ制御レジスタのリセットベクタ キャッチ制御ビット (VC_CORERESET) を使用して、コアがリセットから脱したときにデバッグイベントを生成できます。

ローカルリセットは、P.B1-15「*リセット*」に示されているようにアプリケーションまたはデバッガにより生成され、プロセッサのデバッグ状態を終了して次の動作を引き起こします。

- リセット時に C_HALT および C_DEBUGEN がアサートされている場合、デバッグホールト制御およびステータスレジスタ (DHCSR) の S_HALT および S_REGRDY がセットされ、コアはリセットシーケンスの直後にデバッグ状態に入ります。
- デバッグ例外およびモニタ制御レジスタ (DEMCR) の MON_xxx ビットはクリアされます。

注

1. ARMv7-M には、パワーオンリセットをデバッグする方法はありません。
2. ARMv7-M には、パワーオンリセットとローカルリセットを区別する方法はありません。

DAP で推奨される外部インタフェースで記述されているデバッグロジック リセットと電力制御信号のサポートは実装定義です。

C1.5 デバッグイベントの動作

デバッグ上の理由でトリガされるイベントを、デバッグイベントと呼びます。デバッグイベントにより、次のいずれかの動作が引き起こされます。

- デバッグ状態の開始。ホールトデバッグが有効な（DHCSR の C_DEBUGEN がセットされている）場合、イベントがキャプチャされたときに、プロセッサはデバッグ状態で停止します。
- DebugMonitor 例外。ホールトデバッグが禁止されて（C_DEBUGEN がクリアされている）いて、デバッグモニタが許可されている（DEMCR の MON_EN がセットされている）場合、DebugMonitor のグループ優先度が現在アクティブなグループ優先度よりも高ければ、デバッグイベントにより DebugMonitor 例外が発生します。

  DebugMonitor のグループ優先度が、現在アクティブなグループ優先度と同じか、またはより低い場合、BKPT 命令は HardFault に昇格され、他のデバッグイベント（ウォッチポイントおよび外部デバッグ要求）は無視されます。

  注

  ソフトウェアは、この条件が当てはまる場合、および DebugMonitor 例外が禁止されている場合に、DebugMonitor 例外を保留状態に置くことができます。

- HardFault 例外。ホールトデバッグおよびモニタの両方が禁止されている場合、BKPT 命令は HardFault に昇格され、他のデバッグイベント（ウォッチポイントおよび外部デバッグ要求）は無視されます。

  注

  HardFault またはロックアップを引き起こす BKPT 命令は、回復不能と見なされます。

デバッグフォルト ステータスレジスタ (DFSR) には、キャプチャされる各デバッグイベントに対応するステータスビットが含まれています。これらのビットは、1 を書き込んでクリアです。これらのビットは、デバッグイベントによりプロセッサが停止した場合、または例外が生成された場合にセットされます。イベントが無視された場合にビットが更新されるかどうかは実装定義です。

ホールトおよびデバッグモニタのサポートの要約を表 C1-4 に示します。

表 C1-4 デバッグ関連のフォルト

フォルトの原因	例外サポート（ホールトおよび DebugMonitor）	DFSR ビット名	注
内部停止要求	はい	HALTED	ステップコマンド、コア停止要求など
ブレークポイント	はい	BKPT	BKPT 命令または FPB での一致によるブレークポイント

表 C1-4 デバッグ関連のフォルト

フォルトの原因	例外サポート（ホールトおよびDebugMonitor）	DFSR ビット名	注
ウォッチポイント[a]	はい	DWTTRAP	DWT でのウォッチポイント一致
ベクタキャッチ	ホールトのみ	VCATCH	DEMCR.VC_xxx ビットがセットされる
外部	はい	EXTERNAL	EDBGRQ ラインがアサートされる

a. PC 一致のウォッチポイントを含みます。

ベクタキャッチ機能の説明については、P.C1-16「*ベクタキャッチのサポート*」を参照して下さい。

C1.5.1 デバッグステップ実行

ARMv7-M は、ホールトデバッグとモニタデバッグの両方で、デバッグステップ実行をサポートしています。デバッグ状態からのステップ実行は、デバッグホールト制御およびステータスレジスタ (DHCSR) の C_STEP および C_HALT 制御ビットへの書き込みによりサポートされます。

C_STEP がセットされていて、同じまたはそれ以後のレジスタへの書き込みで C_HALT がクリアされた場合、システムは次の動作を行います。

1. デバッグ状態を終了します。
2. 次のいずれかの動作を実行します。
   - *次の命令*が実行（ステップ実行）されます。
   - 例外開始シーケンスが発生し、次の命令コンテキストをスタックに保存します。プロセッサは、例外の優先度と後着規則に従って例外ハンドラを開始し、最初の命令で停止します。
   - 次の命令が実行（ステップ実行）され、例外モデルにより、期待していたプログラムフローからの変更が引き起こされます。
     — 例外の優先度と後着規則に従って、例外開始シーケンスが発生します。プロセッサは、取得された例外ハンドラの最初の命令を実行する準備ができた時点で停止します。
     — 実行された命令が例外からの復帰命令の場合、テイルチェインにより新しい例外ハンドラが開始されることがあります。プロセッサは、取得された例外ハンドラの最初の命令を実行する準備ができた時点で停止します。

   注

   例外開始の動作は再帰的ではありません。ステップシーケンス内では、PushStack() の更新は 1 回だけ発生できます。
3. デバッグ状態に復帰します。

デバッガは、オプションとして DHCSR の C_MASKINTS ビットをセットし、PendSV、SysTick、および外部の構成可能な割り込みが発生することを禁止（マスク）できます。C_MASKINTS がセットされている場合、許可されている割り込みハンドラがアクティブになり、ステップ実行されている命令とともに実行されます。ステップ実行の制御の要約を表 C1-5 に示します。

表 C1-5 DHSCR を使用したデバッグステップ実行の制御

DHCSR への書き込み[a]			
C_HALT	C_STEP	C_MASKINTS	動作
0	0	0	デバッグ状態を終了し、命令の実行を開始します。 例外の構成規則に従い、例外がアクティブになります。
0	0	1	デバッグ状態を終了し、命令の実行を開始します。 PendSV、SysTick、および外部の構成可能な割り込みは禁止されます。それ以外の場合、標準の構成規則に従って例外がアクティブになります。
0	1	0	デバッグ状態を終了し、命令をステップ実行して停止します。 例外の構成規則に従い、例外がアクティブになります。
0	1	1	デバッグ状態を終了し、命令をステップ実行して停止します。 PendSV、SysTick、および外部の構成可能な割り込みは禁止されます。それ以外の場合、標準の構成規則に従って例外がアクティブになります。
1	x	x	デバッグ状態を維持します。

a. 書き込みが発生したとき、C_DEBUGEN == 1 かつ S_HALT == 1 であることを想定しています（システムが停止している）。

ホールトデバッグのサポートが許可されている（C_DEBUGEN == 1、S_HALT == 0）状態で、システムが実行中に C_STEP または C_MASKINTS を変更した場合、結果は予測不能です。C_DEBUGEN == 0 のとき、C_HALT、C_STEP、C_MASKINTS はハードウェアでは無視され、ソフトウェアからは UNKNOWN です。

注

デバッグ状態で C_HALT がクリアされた場合、以後に S_HALT == "1" が読み出されると、新しいデバッグイベントの検出によりデバッグ状態が再開されたことを意味します。

デバッグモニタのステップ実行

デバッグモニタによるステップ実行は、デバッグ例外およびモニタ制御レジスタの MON_STEP 制御ビットによりサポートされます。MON_STEP がセットされている（同時に C_DEBUGEN がクリアされている）場合、ステップ実行要求は保留中の要求となり、DebugMonitor ハンドラからデバッグ対象のコード（デバッグターゲットのコード）へ復帰したときにアクティブになります。

注

モニタステップ実行要求がアクティブになる前に、テイルチェインにより他の例外ハンドラが実行されることがあります。

ステップ実行要求がアクティブになると、次のステップ実行フェーズのいずれかが実行されます。

- *次の命令*が実行（ステップ実行）されます。
- 例外開始シーケンスが発生し、次の命令コンテキストをスタックに保存します。プロセッサは、例外の優先度と後着規則に従って例外ハンドラを開始し、最初の命令で停止します。
- *次の命令*が実行（ステップ実行）され、例外モデルにより、期待していたプログラムフローからの変更が引き起こされます。
  - 例外の優先度と後着規則に従って、例外開始シーケンスが発生します。プロセッサは、例外処理を開始した例外ハンドラの最初の命令を実行する準備ができた時点で停止します。
  - 実行された命令が例外からの復帰命令の場合、テイルチェインにより新しい例外ハンドラが開始されることがあります。プロセッサは、取得された例外ハンドラの最初の命令を実行する準備ができた時点で停止します。

注

ステップ実行フェーズでは、例外開始シーケンスが再帰的でないため、スタックの更新は 1 回しか行えません。

ステップ実行フェーズの後で、DebugMonitor 例外が取得され、DFSR.HALTED ビットがセットされます。

C1.6 SCS でのデバッグレジスタのサポート

システム制御ブロックのデバッグ機構は、割り込み制御状態レジスタ (ICSR) とデバッグフォルト ステータスレジスタに存在する、2 つのハンドラ関係のフラグビット（ISRPREEMPT および ISRPENDING）で構成されます。

デバッグ制御ブロックに組み込まれている追加デバッグレジスタの概要を表 C1-6 に示します。

表 C1-6 SCS のデバッグレジスタ領域

アドレス	R/W	機能
0xE000EDF0	R/W	デバッグホールト制御およびステータスレジスタ (DHCSR)
0xE000EDF4	WO	デバッグコアレジスタ セレクタレジスタ (DCRSR)
0xE000EDF8	R/W	デバッグコアレジスタ データレジスタ (DCRDR)
0xE000EDFC	R/W	デバッグ例外およびモニタ制御レジスタ (DEMCR)
0xE000EEFF まで	…	デバッグ拡張用に予約

レジスタの詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

C1.6.1 ベクタキャッチのサポート

ベクタキャッチのサポートは、特定の例外が発生したときにデバッグイベントを生成し、デバッグ状態を開始するために使用される機構です。ベクタキャッチは、ホールトデバッグでのみサポートされています。

DHCSR の C_DEBUGEN がセットされ、DEMCR の最低 1 つの VC* イネーブルビットがセットされ、関連付けられている例外がアクティブになった場合、デバッグイベントが発生します。これによって、例外ハンドラの最初の命令でデバッグ状態が開始されます（実行が停止します）。

注

フォルトステータス ビットは例外の開始時にセットされ、エラーの原因を判断するためにデバッガが使用できます。

C1.7 計装トレースマクロセル (ITM) のサポート

計装トレースマクロセル (ITM) は、アプリケーションがログ出力 / イベントのワードをオプションの外部インタフェース (TPIU) へ書き込むための、レジスタベースのインタフェースを提供します。ITM は、タイムスタンプ情報パケットの制御と生成もサポートします。

イベントワードとタイムスタンプ情報はパケットに形成され、データウォッチポイントおよびトレース (DWT) ブロックからのハードウェア イベントパケットと多重化されます。

C1.7.1 動作の理論

ITM は、次の要素で構成されます。

- スティミュラス（スティミュラスポート）レジスタ
- スティミュラス許可（トレース許可）レジスタ
- スティミュラスアクセス（トレース特権）レジスタ
- 汎用制御（トレース制御）レジスタ

スティミュラスポート レジスタの数は、8 の倍数で実装定義です。トレース特権レジスタのビットにすべて 1 を書き込んでから読み出しを行い、いくつのビットがセットされているかを調べることで、サポートされているスティミュラスポートの数を判定できます。

トレース特権レジスタは、関連付けられているスティミュラスポート（8 つごとのグループ）と、対応するトレース許可レジスタのビットが非特権（ユーザ）アクセスによって書き込み可能かどうかを定義しています。ユーザコードは、常にスティミュラスポートを読み出すことができます。

スティミュラスポート レジスタは 32 ビットのレジスタで、ワードアラインの（アドレス [1:0] == 0b00）バイト（ビット [7:0]）、ハーフワード（ビット [15:0]）、またはワードアクセスをサポートしています。非ワードアラインのアクセスの結果は予測不能です。制御レジスタと追加のマスクビットにはグローバルな許可を示す ITMENA があり、トレース許可レジスタ内でスティミュラスポート レジスタを個別に許可します。

許可されているスティミュラスポートへ書き込みが行われた場合、ポートの ID、書き込みアクセスのサイズ、および書き込まれたデータはスティミュラスポートの FIFO へコピーされ、TPIU 経由で送信されます。

FIFO がフルの場合、スティミュラスポートへの書き込みは無視されます。スティミュラスポートの読み出しは、ポートの FIFO ステータスを示します。ITMENA またはスティミュラスポートのイネーブルビットがクリア（ポートが禁止されている）の場合、FIFO ステータスを読み出すと FULL が返されます。ITMENA はパワーオンリセット時にクリアされます。

注

システムの正常性を保証するため、ソフトウェアポーリング方式で排他アクセスを使用し、スティミュラスポートの FIFO ステータスに対応するスティミュラスポートへの書き込みを管理できます。ソフトウェアは、書き込む先のスティミュラスポートから読み出しを行い、FIFO のステータスをテストする必要があります。

すべての ITM レジスタは、非特権（ユーザ）および特権コードから、いつでも読み出し可能です。特権書き込みアクセスは、ITMENA がセットされていない限り無視されます。トレース制御およびトレース特権レジスタへの非特権書き込みアクセスは、常に無視されます。スティミュラスポートおよびトレース許可レジスタへの非特権書き込みアクセスは、トレース特権レジスタの設定に従い、許可または無視されます。トレース許可レジスタは、トレース特権レジスタの設定に従い、バイト単位でユーザアクセスが許可されます。

タイムスタンプのサポート

タイムスタンプは、トレース出力ポートでの観測という形でイベント生成のタイミング情報を提供します。タイムスタンプカウンタのサイズとクロック周波数は実装定義です。

C1.7.2 ITM のレジスタサポート

レジスタの詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

C1.8 データウォッチポイントおよびトレース (DWT) のサポート

データウォッチポイントおよびトレース (DWT) コンポーネントは、次の機能を提供します。

- PC サンプリング -2 つの形式がサポートされています。
  - — DWT 機能一致の結果としてのデータトレース出力
  - — PC サンプルレジスタを使用した外部 PC サンプリング
- コンパレータは次の機能をサポートしています。
  - — ウォッチポイント - デバッグ状態の開始または DebugMonitor 例外の取得
  - — データトレース
  - — ETM などの外部リソースで使用するためのシグナル出力
  - — サイクルカウント一致
- 例外トレースのサポート。
- プロファイリングカウンタのサポート。

C1.8.1 動作の理論

例外トレースと外部エージェントの PC サンプリング機能（プログラムカウンタ サンプルレジスタのサポート）を除いて、DWT の機能はカウンタまたはコンパレータをベースとしています。サポートされている機能は、デバッグ ROM テーブル、DEMCR.TRCENA マスタイネーブルビット、および DWT 制御レジスタ (DWT_CTRL) 内の機能有効ビットから判定できます。詳細については、P.C1-9「*ARMv7-M でのデバッグサポート*」を参照して下さい。例外トレースとカウンタ制御は、DWT_CTRL レジスタにより提供されます。ウォッチポイントおよびデータトレースサポートは、一組の比較、マスク、機能レジスタ（DWT_COMPx、DWT_MASKx、DWT_FUNCTIONx）を使用します。

DWT で生成されたイベントは、3 つの動作のいずれかを引き起こします。

- ハードウェアソース パケットの生成。パケットが生成され、他のイベント、制御、タイムスタンプパケットと結合されます。
- コアの停止 - デバッグ状態が開始されます。
- DebugMonitor 例外
- 外部デバッグリソースへの制御入力としての CMPMATCH[N] 信号の生成

注

DWT ハードウェアイベント パケットの伝送は、ITM ブロック（トレース制御レジスタ）で許可されます。このブロックは、タイムスタンプサポートも制御します。タイプスタンプ機構については、ITM スティミュラスポートを禁止できます。ただし、タイムスタンプ機能を提供するため、ITM トレース制御レジスタの ITMENA および TSENA はセットされている必要があります。

例外トレースのサポート

例外トレースは、DWT_CTRL レジスタの EXCTRCENA ビットを使用して許可されます。このビットがセットされている場合、DWT は次の条件で例外トレースパケットを送信します。

- 例外の開始時（スレッドモードから、またはスレッドやハンドラの横取り）
- ハンドラが EXC_RETURN ベクタで終了した場合の例外終了時
- 横取りされたスレッドまたはハンドラコード シーケンスを再開する例外復帰時

プログラムカウンタのサンプリングのサポート

DWT プログラムカウンタ サンプリングレジスタ (DWT_PCSR) は ARMv7-M の実装定義オプションです。このレジスタは、デバイスで現在実行中のコードの動作を変更せず、デバッガからアクセス可能であるように定義されています。これによって、コアで実行中のコードの粒度の粗い非侵襲性のプロファイリングを行う機構が提供されます。DWT_PCSR はワードアクセスが可能な読み出し専用レジスタで、このレジスタへの書き込みは無視されます。バイトまたはハーフワード読み出しの結果は予測不能です。レジスタが読み出されると、次のいずれかが返されます。

- コアによって*最近実行された*命令のアドレス
- 実装されており、プロセッサがデバッグ状態、または非侵襲性のデバッグが許可されていない状態とモードの場合、0xFFFFFFFF
- 実装されていない場合、RAZ

注

*最近実行された*の意味については、アーキテクチャで定義されていません。コアによって実行されている命令と、DWT_PCSR に現れるアドレスとの間の遅延は定義されていません。例えば、コードの一部が実行中に、自分が実行されているプロセッサの DWT_PCSR を読み出した場合、コードのその部分に対応するプログラムカウンタと、読み出される値との関係は保証されていません。DWT_PCSR は、外部エージェントがコードプロファイリングのための統計情報を提供するために使用することのみを意図しています。ARM コアによって直接 DWT_PCSR へ読み出しアクセスを行った場合、返される値は未知です。

デバッグエージェントは、値として 0xFFFFFFFF が返された場合、コアが停止していることを示していると前提しないようにする必要があります。この目的には、デバッグホールト制御およびステータスレジスタの S_HALT ビットを使用します。

DWT_PCSR は、DWT 制御レジスタの PCSAMPLENA ビットの影響を受けません。

C1.8.2 DWT のレジスタサポート

レジスタの詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

C1.9 エンベデッドトレース (ETM) のサポート

ETM は ARMv7-M のオプション機能です。この機能がサポートされている場合、ETM および DWT/ITM パケットソースからの出力パケットストリームのフォーマットを行うことができる TPIU ポートが準備されている必要があります。

ETM サポートの詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

C1.10 トレースポート インタフェースユニット (TPIU)

DWT ブロックからのハードウェアイベントと、ITM ブロックからのソフトウェアイベントは、タイムスタンプ情報とともにパケットストリームへ多重化されます。タイプスタンプ情報およびパケットストリームの制御および構成は、DWT および ITM ブロックの一部です。パケットが可視にされる（ピンまたはトレースバッファとアクセス機構を提供する必要があります）か、コア内で完結するかは実装定義です。

直接の可視性を実現するには、実装でトレースポート インタフェースユニット (TPIU) を提供する必要があります。ARMv7-M TPIU のプログラマモデルには、非同期シリアルワイヤ出力 (SWO) または同期（単一または複数ビットのデータパス）トレースポートのサポートが含まれています。DWT/ITM パケットストリームと SWO の組み合わせは、シリアルワイヤ ビューア (SWV) と呼ばれます。

ARMv7-M の最小の TPIU サポートでは、ハードウェアやソフトウェアにより生成されたイベント情報の、DWT/ITM により生成されるパケットストリームの出力パスが提供されます。これは、パススルーモードで動作している TPIU での、デバッグトレースに対する TPIU サポートと呼ばれます。

TPIU サポートの詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

C1.11 フラッシュパッチとブレークポイント (FPB) のサポート

フラッシュパッチとブレークポイント (FPB) コンポーネントは、次の機能をサポートできます。

- システムメモリに含まれるコード領域の特定のリテラル位置を、SRAM 領域のアドレスにリマッピングする。[1]
- システムメモリに含まれるコード領域の特定の命令位置を、SRAM 領域のアドレスにリマッピングする。[1]
- 命令フェッチに対するブレークポイント機能

これらの各機能について、FPB のサポートは実装定義です。

注

FPB は、デバッグでの使用に限定されてはいません。FPB は通常のコード実行条件下でも同様に動作するため、製品の更新サポートにも使用できます。

C1.11.1 動作の理論

レジスタには、次の 3 つのタイプがあります。

- 汎用制御レジスタ FP_CTRL
- リマップアドレス レジスタ FP_REMAP
- フラッシュパッチコンパレータ レジスタ

命令の比較とリテラルの比較には、独立したコンパレータが使用されます。それぞれの個数は実装定義で、FP_CTRL レジスタから読み出せます。

命令一致フラッシュパッチコンパレータ レジスタは、命令をリマップするか、またはブレークポイントを生成するように構成できます。

リテラル一致コンパレータの機能は固定されており、データ読み出しアクセスのリマッピング機能のみをサポートします。読み出し時のリテラル一致は、データのワード、ハーフワード、またはバイト単位で行うことができます。一致が起きると、リマッピングされた位置から適切なデータがフェッチされます。

次の制限が適用されます。

- リマッピングの影響を受けるアンアラインドのリテラルアクセスは、実装定義です。
- MPU が許可されている場合、元のアドレスで MPU チェックが実行され、リマッピングされた位置に属性が適用されます。リマッピングされたアドレスは、MPU でチェックされません。
- 排他ロードアクセスはリマッピング可能ですが、排他アクセスとして実行されるかどうかは予測不能です。

1. アドレス領域の詳細については、P.B2-3 表 B2-1 を参照して下さい。

- ブレークポイントとして構成されている 32 ビット命令での命令一致は、命令の最初のハーフワード、または両方のハーフワードが一致するように構成する必要があります。32 ビット命令の 2 番目のハーフワードのアドレスにのみ一致するブレークポイントがデバッグイベントを生成するかどうかは予測不能です。

各コンパレータには独自のイネーブルビットがあり、グローバル イネーブルビットがセットされているときに有効になります。

C1.11.2 FPB のレジスタサポート

レジスタの詳細については、該当する ARM コアまたはデバイスのドキュメントを参照して下さい。

パート D

付録

付録 A
CPUID

ARMv7-M で使用されている CPUID 方式は、改訂フォーマットの ARM アーキテクチャ CPUID 方式に適合しています。メイン ID レジスタ (CPUID[:19:16]) でアーキテクチャバリアントとして 0xF を指定していることで、改訂フォーマットを使用していることを示しています。すべての ID レジスタは、特権アクセスのみが許可されます。特権書き込みは無視され、非特権データアクセスは BusFault エラーを引き起こします。

A.1 コア機能 ID レジスタ

コア機能IDレジスタは、システム制御空間で、表 A-1で定義されているようにデコードされます。

表 A-1 SCS でのコア機能 ID レジスタのサポート

アドレス	タイプ	リセット時の値	機能
0xE000ED00	読み出し専用	実装定義	CPUID ベースレジスタ
0xE000ED40	読み出し専用	実装定義	PFR0：プロセッサ機能レジスタ 0
0xE000ED44	読み出し専用	実装定義	PFR1：プロセッサ機能レジスタ 1
0xE000ED48	読み出し専用	実装定義	DFR0：デバッグ機能レジスタ 0
0xE000ED4C	読み出し専用	実装定義	AFR0：補助機能レジスタ 0
0xE000ED50	読み出し専用	実装定義	MMFR0：メモリモデル機能レジスタ 0
0xE000ED54	読み出し専用	実装定義	MMFR1：メモリモデル機能レジスタ 1
0xE000ED58	読み出し専用	実装定義	MMFR2：メモリモデル機能レジスタ 2
0xE000ED5C	読み出し専用	実装定義	MMFR3：メモリモデル機能レジスタ 3
0xE000ED60	読み出し専用	実装定義	ISAR0：ISA 機能レジスタ 0
0xE000ED64	読み出し専用	実装定義	ISAR1：ISA 機能レジスタ 1
0xE000ED68	読み出し専用	実装定義	ISAR2：ISA 機能レジスタ 2
0xE000ED6C	読み出し専用	実装定義	ISAR3：ISA 機能レジスタ 3
0xE000ED70	読み出し専用	実装定義	ISAR4：ISA 機能レジスタ 4
0xE000ED74	読み出し専用	実装定義	ISAR5：ISA 機能レジスタ 5
0xE000ED78	読み出し専用	実装定義	予約 (RAZ)
0xE000ED7C	読み出し専用	実装定義	予約 (RAZ)

バージョンフィールドの 2 つの値は、特別な意味を持ちます。

Field[ ] == すべて 0

このデバイスには、該当機能が存在しないか、あるいはフィールドが割り当てられていません。

Field[ ] == すべて 1

そのフィールドではオーバフローしたので、ID 空間の他の場所で定義されています。

注

コア機能 ID レジスタの予約フィールドは、すべて 0 として読み出されます (RAZ) 。
属性レジスタの詳細については、該当する ARM 準拠コアのテクニカル リファレンスマニュアルを参照して下さい。

付録 B
過去の命令のニーモニック

次の表は、各命令に対する Thumb-2 や等価な UAL 構文が導入される前の Thumb 命令で使用されていた、UAL 以前のアセンブリ構文です。この表は、正しくアセンブル可能な UAL 以前の Thumb アセンブラコードを、UAL アセンブラコードに変換するために使用することができます。

この表は、UAL アセンブラコードから UAL 以前の Thumb アセンブラコードへの逆変換に使用することを目的としたものではありません。

UAL 以前の Thumb アセンブラにより選択されたものと同じ命令エンコードが UAL アセンブラでも選択されることを保証するために、2 オペランド形式を使用する必要がある 1 つの場合を除いて、この表では等価な UAL 構文の 3 オペランド形式が使用されています。

表 B-1 UAL 以前のアセンブリ構文

UAL 以前の Thumb 構文	等価な UAL 構文	注
ADC <Rd>, <Rm>	ADCS <Rd>, <Rd>, <Rm>	
ADD <Rd>, <Rn>, #<imm>	ADDS <Rd>, <Rn>, #<imm>	
ADD <Rd>, #<imm>	ADDS <Rd>, #<imm>	
ADD <Rd>, <Rn>, <Rm>	ADDS <Rd>, <Rn>, <Rm>	
ADD <Rd>, SP	ADD <Rd>, SP, <Rd>	

表 B-1 UAL 以前のアセンブリ構文

UAL 以前の Thumb 構文	等価な UAL 構文	注
ADD <Rd>, <Rm>	ADD <Rd>, <Rd>, <Rm>	<Rd> または <Rm> が上位レジスタで、<Rm> が SP ではない場合
ADD <Rd>, PC, #<imm> ADR <Rd>, <label>	ADD <Rd>, PC, #<imm> ADR <Rd>, <label>	可能な場合は ADR 形式を推奨
ADD <Rd>, SP, #<imm>	ADD <Rd>, SP, #<imm>	
ADD SP, #<imm>	ADD SP, SP, #<imm>	
AND <Rd>, <Rm>	ANDS <Rd>, <Rd>, <Rm>	
ASR <Rd>, <Rm>, #<imm>	ASRS <Rd>, <Rm>, #<imm>	
ASR <Rd>, <Rs>	ASRS <Rd>, <Rd>, <Rs>	
B<cond> <label>	B<cond> <label>	
B <label>	B <label>	
BIC <Rd>, <Rm>	BICS <Rd>, <Rd>, <Rm>	
BKPT <imm>	BKPT <imm>	
BL <label>	BL <label>	
BLX <Rm>	BLX <Rm>	<Rm> には上位レジスタも使用可能です。
BX <Rm>	BX <Rm>	<Rm> には上位レジスタも使用可能です。
CMN <Rn>, <Rm>	CMN <Rn>, <Rm>	
CMP <Rn>, #<imm>	CMP <Rn>, #<imm>	
CMP <Rn>, <Rm>	CMP <Rn>, <Rm>	<Rd> と <Rm> には上位レジスタも使用可能です。
CPS<effect> <iflags>	CPS<effect> <iflags>	
CPY <Rd>, <Rm>	MOV <Rd>, <Rm>	
EOR <Rd>, <Rm>	EORS <Rd>, <Rd>, <Rm>	
LDMIA <Rn>!, <registers>	LDMIA <Rn>, <registers> LDMIA <Rn>!, <registers>	<Rn> が <registers> のリストに含まれている場合 それ以外の場合
LDR <Rd>, [<Rn>, #<imm>]	LDR <Rd>, [<Rn>, #<imm>]	<Rn> には SP も使用可能です。

表 B-1 UAL 以前のアセンブリ構文

UAL 以前の Thumb 構文	等価な UAL 構文	注
LDR <Rd>, [<Rn>, <Rm>]	LDR <Rd>, [<Rn>, <Rm>]	
LDR <Rd>, [PC, #<imm>] LDR <Rd>, <label>	LDR <Rd>, [PC, #<imm>] LDR <Rd>, <label>	可能な場合は <label> 形式を推奨
LDRB <Rd>, [<Rn>, #<imm>]	LDRB <Rd>, [<Rn>, #<imm>]	
LDRB <Rd>, [<Rn>, <Rm>]	LDRB <Rd>, [<Rn>, <Rm>]	
LDRH <Rd>, [<Rn>, #<imm>]	LDRH <Rd>, [<Rn>, #<imm>]	
LDRH <Rd>, [<Rn>, <Rm>]	LDRH <Rd>, [<Rn>, <Rm>]	
LDRSB <Rd>, [<Rn>, <Rm>]	LDRSB <Rd>, [<Rn>, <Rm>]	
LDRSH <Rd>, [<Rn>, <Rm>]	LDRSH <Rd>, [<Rn>, <Rm>]	
LSL <Rd>, <Rm>, #<imm>	LSLS <Rd>, <Rm>, #<imm>	
LSL <Rd>, <Rs>	LSLS <Rd>, <Rd>, <Rs>	
LSR <Rd>, <Rm>, #<imm>	LSRS <Rd>, <Rm>, #<imm>	
LSR <Rd>, <Rs>	LSRS <Rd>, <Rd>, <Rs>	
MOV <Rd>, #<imm>	MOVS <Rd>, #<imm>	
MOV <Rd>, <Rm>	ADDS <Rd>, <Rm>, #0 MOV <Rd>, <Rm>	<Rd> と <Rm> が両方とも R0 ～ R7 の場合 それ以外の場合
MUL <Rd>, <Rm>	MULS <Rd>, <Rm>, <Rd>	
MVN <Rd>, <Rm>	MVNS <Rd>, <Rm>	
NEG <Rd>, <Rm>	RSBS <Rd>, <Rm>, #0	
ORR <Rd>, <Rm>	ORRS <Rd>, <Rd>, <Rm>	
POP <registers>	POP <registers>	<registers> には PC を含めることが可能
PUSH <registers>	PUSH <registers>	<registers> には LR を含めることが可能
REV <Rd>, <Rn>	REV <Rd>, <Rn	
REV16 <Rd>, <Rn>	REV16 <Rd>, <Rn	
REVSH <Rd>, <Rn>	REVSH <Rd>, <Rn	
ROR <Rd>, <Rs>	RORS <Rd>, <Rd>, <Rs>	

表 B-1 UAL 以前のアセンブリ構文

UAL 以前の Thumb 構文	等価な UAL 構文	注
SBC <Rd>, <Rm>	SBCS <Rd>, <Rd>, <Rm>	
STMIA <Rn>!, <registers>	STMIA <Rn>!, <registers>	
STR <Rd>, [<Rn>, #<imm>]	STR <Rd>, [<Rn>, #<imm>]	<Rn> には SP も使用可能
STR <Rd>, [<Rn>, <Rm>]	STR <Rd>, [<Rn>, <Rm>]	
STRB <Rd>, [<Rn>, #<imm>]	STRB <Rd>, [<Rn>, #<imm>]	
STRB <Rd>, [<Rn>, <Rm>]	STRB <Rd>, [<Rn>, <Rm>]	
STRH <Rd>, [<Rn>, #<imm>]	STRH <Rd>, [<Rn>, #<imm>]	
STRH <Rd>, [<Rn>, <Rm>]	STRH <Rd>, [<Rn>, <Rm>]	
SUB <Rd>, <Rn>, #<imm>	SUBS <Rd>, <Rn>, #<imm>	
SUB <Rd>, #<imm>	SUBS <Rd>, #<imm>	
SUB <Rd>, <Rn>, <Rm>	SUBS <Rd>, <Rn>, <Rm>	
SUB SP, #<imm>	SUB SP, SP, #<imm>	
SWI <imm>	SVC <imm>	
SXTB <Rd>, <Rm>	SXTB <Rd>, <Rm>	
SXTH <Rd>, <Rm>	SXTH <Rd>, <Rm>	
TST <Rn>, <Rm>	TST <Rn>, <Rm>	
UXTB <Rd>, <Rm>	UXTB <Rd>, <Rm>	
UXTH <Rd>, <Rm>	UXTH <Rd>, <Rm>	

付録 C
非推奨の ARMv7-M 機能

Thumb 命令セットのいくつかの機能は、ARMv7 では推奨されていません。ARMv7-M でサポートされている命令に影響する非推奨の機能は次のとおりです。

- 16 ビット ADD（SP + レジスタ）命令で、PC を Rd または Rm として使用
- 16 ビット ADD（SP + レジスタ）命令で、SP を Rm として使用
- 16 ビット CMP（レジスタ）命令で、SP を Rm として使用
- <Rd> が SP または PC で、かつ <Rm> が同様に SP または PC である MOV（レジスタ）命令の使用
- 16 ビット STM 命令のレジスタリストで、Rn を最下位のレジスタとして、ベースレジスタライトバックとともに使用

次に挙げる ARMv7-M の追加機能は非推奨です。

- 4 バイトアラインド スタックのサポート (CCR.STKALIGN == "0")

付録 D
擬似コードの定義

この付録では、本書で使用されている擬似コードの正式な定義を示します。また、擬似コードが使用する、有用なアーキテクチャ固有のジョブを実行するためのヘルパプロシージャおよび関数のリストも示します。本章は以下のセクションから構成されています。

D.1 命令エンコード図および擬似コード

本書における命令の説明には以下が含まれています。

- エンコードセクションには 1 つ以上のエンコード図が含まれ、それぞれの図に続いて、エンコードのフィールドを命令の共通擬似コードの入力に変換するエンコード固有の擬似コードがあり、そのエンコード固有の特殊ケースを示します。
- 動作セクションには、説明されているそのエンコードすべてに適用される共通の擬似コードが含まれています。動作セクションの擬似コードには、開始時、または EncodingSpecificOperations() により条件チェックのみが実行された直後に、if ConditionPassed() then 関数への呼び出しが含まれます。

エンコード図では、命令の各ビットが次のいずれかとして指定されます。

- 必ず 0 または 1 であるビット。図の中では 0 または 1 として表されます。このビットがこの値を持たない場合、エンコードは別の命令に対応します。
- 0 または 1 の*必要があるビット*。図の中では (0) または (1) として表されます。このビットがこの値を持たない場合、命令は予測不能です。
- 名前付きのビット、または名前付きのマルチビットフィールドに含まれている 1 ビット。

必ず 0 または 1 でなければならないすべてのビットがエンコード図と命令とで一致する場合、そのエンコード図はその命令と一致します。

命令の実行モデルは次のとおりです。

1. その命令と一致するすべてのエンコード図を見つけます。一致するエンコード図が無いこともあります。その場合、その実行モデルを破棄して、代わりに、どのように命令が取り扱われるかを調べるために関連する命令セットの章を参照して下さい（その他の可能性もありますが、通常はそのような命令のビットパターンは、予約されていて未定義です。例えば、未割り当てのヒント命令は予約されていて、NOP として実行されると記載されています）。

2. 一致するエンコード図の動作擬似コードが条件チェックから始まる場合、その条件チェックを実行します。条件チェックが失敗した場合は、その実行モデルを破棄して、命令を NOP として取り扱います（複数の一致するエンコード図が存在する場合、それらに対応する共通擬似コードのすべてが条件チェックから開始されるか、またはいずれの擬似コードも開始されません）。

3. 一致する各エンコード図について、個別にかつ平行して、エンコード固有の擬似コードを実行します。これらのエンコード固有の擬似コードは、対応するエンコード図の、それぞれの名前付きビットまたはマルチビットフィールドのビットストリング変数で始まります。これらの変数には、ビットまたはマルチビットフィールドと同じ名前が付けられていて、命令のビットパターンに含まれている、対応するビットの値で初期化されます。

   エンコード図に、同じ名前のビットまたはフィールドが複数含まれている場合もあります。この場合、これらのビットまたはフィールドは、すべて同一の値であることが前提です。エンコード固有の擬似コードには、これらの値が同一ではない場合にどのような動作が行われるかが、Consistent() 関数を使って示されています。命令に含まれている、引数と同じ名前のすべてのビットまたはフィールドが同じ値を持っている場合、この関数は TRUE を返します。違う場合は FALSE を返します。

一致するエンコード図が複数存在する場合、それらに対応する擬似コードのうち、1 つを除くすべてに、それが適用されないことを示す特別な場合が含まれている必要があります。それらの擬似コードと、対応するエンコード図の結果をすべて破棄します。

これによって、1 つの擬似コードとそれに対応するエンコード図が考慮の対象として残ります。この擬似コードも特別な場合を含んでいる可能性があります（ほとんどの場合、これが予測不能であることを示すものです）。その場合は、この実行モデルを破棄して、命令をその特別な場合に従って取り扱います。

4. エンコード図の、*0 または 1 の必要があるビット*を、命令のビットパターンの対応するビットと照合します。一致しないビットがある場合は、この実行モデルを破棄して、命令を予測不能として取り扱います。

5. エンコード図を含む命令記述の残りの動作擬似コードを実行します。この擬似コードは、エンコード固有の擬似コードにより残された値に設定されたすべての変数を使用して開始されます。

共通の擬似コードでの ConditionPassed() （存在する場合）呼び出しは、手順 2 を実行します。EncodingSpecificOperations() 呼び出しは、手順 3 と 4 を実行します。

D.1.1 擬似コード

擬似コードは、命令が実行する動作を正確に記述したものです。命令フィールドは、命令のエンコード図で示されている名前により参照されます。

擬似コードについては、以下のセクションで詳しく説明します。

D.2 擬似コードの制限

命令の機能の擬似コードによる記述には、いくつか制限があります。これらの制限は主に、明瞭かつ簡潔にするために、擬似コードがシーケンシャルであいまいさのない言語であることが原因です。

制限には次のものが含まれます。

- 擬似コードでは、命令が複数のメモリアクセスを生成する場合、順序の要件が記述されません。メモリアクセス時の順序要件の説明については、P.A3-31「*メモリアクセスの順序*」を参照して下さい。
- 擬似コードでは、未定義の命令が条件チェックに失敗した場合の厳密なルールが記述されていません。そのような場合は、擬似コードの UNDEFINED ステートメントが、直接または EncodingSpecificOperations() 関数の呼び出し内で if ConditionPassed() then ... 構造の中に置かれていることにより、命令が NOP として実行されることが示されます。詳細については、P.A6-9「*未定義命令の条件付き実行*」を参照して下さい。
- 擬似コードの UNDEFINED、UNPREDICTABLE、SEE ステートメントは、実行されている擬似コードによって示されているのとは異なる動作を示しています。これらのステートメントのいずれかが実行されると、次の動作が行われます。
  - 擬似コードにより示されている、それ以前の動作は、そのステートメントが実行されたと判定するに必要な範囲まで動作が進んだものとして指定されます。
  - 擬似コードにより示されている以後の動作は実行されません。つまり、これらのステートメントによって擬似コードの実行は終了します。

  詳細については、P.AppxD-17「*単純な文*」を参照して下さい。

D.3 データタイプ

このセクションでは、次のトピックについて説明します。

D.3.1 データ型の一般的な規則

ARM アーキテクチャの擬似コードは、強く型付けされた言語です。すべての定数と変数は、次のいずれかの型です。

- ビットストリング
- 整数
- ブール
- 実数
- 列挙型
- リスト
- 配列

定数の型はその構文により決定されます。通常、変数の型は、変数への割り当てにより決定されます。これは、変数は割り当てられるデータと同じ型であることが暗黙的に宣言されているためです。例えば、割り当て x = 1, y = '1'、z = TRUE は、変数 x、y、z が、それぞれ整数、長さ 1 のビットストリング、ブールの型であることを暗黙的に宣言しています。

型の名前を変数名の前に置くことにより、変数の型を明示的に宣言することもできます。これはほとんどの場合、関数の定義で引数や結果の型を宣言するために行われます。

これらのデータ型については、以下のセクションで詳しく説明します。

D.3.2 ビットストリング

ビットストリングは、0 および 1 のビットの、有限の長さのストリングです。長さが異なるビットストリングは、異なる型と見なされます。ビットストリングの最小の長さは 1 です。

長さ N のビットストリングの型名は bits(N) です。bits(1) は bit と同義です。

ビットストリング定数は、0 と 1 のストリングを一重引用符 (') で囲った形式で表記されます。例えば、bit 型の 2 つの定数は、'0' および '1' と表記されます。ビットストリングには、分かりやすくするためにスペースを含めることができます。

'x' 個のビットを持つ特別な形式のビットストリング定数は、ビットストリングの比較で許可されます。詳細については P.AppxD-11「*等価と非等価のテスト*」を参照して下さい。

すべてのビットストリングの値は左から右の順序で表現され、ビットは標準的な*リトルエディアン*順序で番号付けされます。つまり、長さ N のビットストリングの最も左のビットはビット N-1、一番右のビットはビット 0 です。この順序は、整数への変換または整数からの変換で、最上位ビットから最下位ビットへの順序として使用されます。ビットストリング定数、およびエンコード図から派生するビットストリングの場合、この順序は印刷されている順序と一致します。

ビットストリングは、レジスタの内容、メモリの位置、命令などと直接対応するという点において、擬似コードにおける唯一の具体的なデータ型です。残りのデータ型はすべて抽象的です。

D.3.3 整数

擬似コードの整数にはサイズの制限がなく、正と負の両方が許可されます。そのため、これらは、コンピュータ言語やアーキテクチャにおける一般的な意味での整数ではなく、数学的な意味の整数です。コンピュータの整数は、擬似コードでは適切な長さのビットストリングとして表され、これらのビットストリングを整数として解釈するための適切な関数と関連付けられています。

整数の型名は integer です。

整数の定数は通常、0、15、-1234 などのように 10 進数で表記されます。0x55 や 0x80000000 のように、C 形式の 16 進数で表記されることもあります。16 進整数の定数は、マイナスの符号が値の前に無い場合は正の値として扱われます。例えば、0x80000000 は整数 $+2^{31}$ です。-2^{31} を 16 進数で表記する必要がある場合は、-0x80000000 と表記します。

D.3.4 実数

擬似コードの実数はサイズと精度に制限がありません。そのため、これらは数学的な実数であり、コンピュータの浮動小数点数とは異なります。コンピュータの浮動小数点数は、擬似コードでは適切な長さのビットストリングとして表され、これらのビットストリングを実数として解釈するための適切な関数と関連付けられています。

実数の型名は real です。

実数の定数は、小数点付きの 10 進数で表記されます（このため、0 は整数の定数ですが、0.0 は実数の定数です）。

D.3.5 ブール

ブールは論理的な TRUE または FALSE の値です。

ブールの型名は boolean です。これは、長さ 1 のビットストリングである bit とは異なる型です。ブールの定数は TRUE と FALSE です。

D.3.6 列挙型

列挙型は、次のようなシンボリックな定数の定義されたセットです。

```
enumeration SRType (SRType_None, SRType_LSL, SRType_LSR,
                    SRType_ASR, SRType_ROR, SRType_RRX) ;
```

列挙型は常に、1 つ以上のシンボリック定数を含みます。シンボリック定数を複数の列挙型で共有することはできません。

列挙型の変数はシンボリック定数の 1 つを割り当てることにより、通常どおり暗黙的に宣言することが可能ですが、列挙型自体は明示的に宣言されている必要があります。慣例により、それぞれのシンボリック定数は、列挙型の名前から始まり、その後に下線が続きます。列挙型の名前は型名で、シンボリック定数はその定数になる可能性のあるものです。

注

ブールは基本的には、事前に宣言された列挙型です。

enumeration boolean {FALSE, TRUE};
この式は、通常の命名規則に従っておらず、if ステートメントなどのいくつかの擬似コード構造で特別な役割を持っています。

D.3.7 リスト

リストは、他のデータアイテムの順序付けされたセットです。これは、次のようにカンマで区切られ、括弧でくくられています。

(bits(32) shifter_result, bit shifter_carry_out)

リストは常に、1 つ以上のデータアイテムを含んでいます。

リストは、複数の結果を戻す関数の戻り型として多く利用されます。例えば、この例のリストは、最初のオペランドの型が Shift_C() のときに、標準的な ARM シフトまたはローテートを実行する、bits(32) 関数の戻り型です。

いくつかの特定の擬似コードの演算子は、通常の括弧ではなく別形式の括弧でくくられたリストを使用します。具体的には、次のものがあります。

- ビットストリング抽出演算子。これは不等号 "<...>" でくくられたビット番号またはビット番号の範囲のリストを使用します。
- 配列のインデクス付け。これは角括弧 "[...]" でくくられた配列インデクスのリストを使用します。
- 関数に配列と似た引数を渡す場合。これは角括弧 "[...]" でくくられた関数の引数のリストを使用します。

リストに含まれているデータ型のそれぞれの組み合わせは個別の型で、型名は単にデータ型をリストするだけで与えられます（上の例の場合、(bits(32) ,bit) です）。一般的な原則として、型は代入により宣言することができますが、この原則はリストの中の個別のリストアイテムにまで広げることができます。次に例を示します。

```
(shift_t, shift_n) = ('00', 0) ;
```

上の式は、shift_t、shift_n および (shift_t,shift_n) が、それぞれ bits(2) 、integer、(bits(2) ,integer) 型であることを暗黙的に宣言しています。

リストの型は、明示的に名前が付けられた要素をリスト中で使って、明示的に指定することもできます。次に例を示します。

```
type ShiftSpec is (bits(2) shift, integer amount) ;
```

この定義と宣言の後に次のように記述します。


```
ShiftSpec abc;
```

結果として作成されるリストの要素は、"abc.shift" および "abc.amount" として参照することができます。このようなリストの要素の修飾名は、明示的に宣言された変数の場合のみ許可されており、代入でのみ宣言されたものでは許可されていません。

型に明示的に名前付けしても、型自体は変更されません。例えば、上記の ShiftSpec 宣言の後で、ShiftSpec および (bits(2) ,integer) は、同じ型を表す 2 つの異なる名前であり、2 つの異なる型の名前ではありません。リストの要素への参照が不明瞭になることを避けるために、型の異なる名前を使ってリスト変数を複数回数宣言したり、宣言に使われた名前と関連していないリストの要素名でリスト変数を修飾したりすると、エラーが発生します。

代入されているリストのアイテムは、代入されるリスト値の対応するアイテムが破棄されることを示すために、"-" と書かれることがあります。次に例を示します。

```
(shifted, -) = LSL_C(operand, amount) ;
```

リスト定数は、上の例にある ('00', 0) のように、適切な型の定数のリストとして記述されます。

D.3.8 配列

擬似コードの配列は、列挙型または整数の範囲（範囲の下限、".."、および範囲の上限により表されます。この範囲には、上限と加減の数値が含まれます）によりインデクス付けされます。次に例を示します。

```
enumeration PhysReg {
    PhysReg_R0,    PhysReg_R1,     PhysReg_R2,    PhysReg_R3,
    PhysReg_R4,    PhysReg_R5,     PhysReg_R6,    PhysReg_R7,
    PhysReg_R8,    PhysReg_R9,     PhysReg_R10,   PhysReg_R11,
    PhysReg_R12,   PhysReg_SP_Process,            PhysReg_SP_Main,
    PhysReg_LR,    PhysReg_PC};

array bits(32) _R[PhysReg];

array bits(8) _Memory[0..0xFFFFFFFF];
```

配列は常に明示的に宣言され、定数配列の表記はありません。配列には常に 1 つ以上の要素が含まれます。これは、列挙型は常に 1 つ以上のシンボリック定数を含み、整数の範囲は常に 1 つ以上の整数を含んでいるためです。

通常、配列は擬似コードには直接出現しません。擬似コードで、構文的に配列のように見えるアイテムは、通常、R[i]、MemU[address,size]、Element[i,type] などの配列様関数です。これらの関数は、レジスタのバンク処理、メモリ保護、エンディアン依存のバイト順序、排他アクセスの保守、ベクタエレメントの処理など、通常は基礎となる配列へのアクセスに対して実行される、抽象的な追加操作をまとめたものです。

D.4 式

このセクションでは、次のトピックについて説明します。

- *一般的な式の構文*
- *演算子と関数- 多相性とプロトタイプ*: P.AppxD-10
- *優先順位の規則*: P.AppxD-10

D.4.1 一般的な式の構文

式は、次のいずれか 1 つの形式です。

- 定数
- 変数。オプションとして、型を宣言するデータ型名が前に記述されることがあります。
- 型を宣言するデータ型名の後にある、UNKNOWN という単語
- 言語定義の演算子を他の式に適用した結果
- 関数を他の式に適用した結果

変数名は通常、英数字と下線で構成され、最初の文字は英字または下線です。

テキストに記述されている各レジスタは、対応した名前を持つビットストリング変数の宣言と見なされ、その変数はそのレジスタに指定されている動作を行います。例えば、レジスタのビットが 0 として読み出され、書き込みが無視されることが指定されている場合、その変数の対応するビットは 0 として読み出され、書き込みは無視されます。

"bits(32) UNKNOWN" のような式は、式の結果が指定された型の値であるが、アーキテクチャでは値が何であるかは指定されていないため、ソフトウェアはこのような値に依存してはならないことを示しています。生成された値がセキュリティホールを構成しないようにする必要があります。また、この値がソフトウェアに有用な情報を提供するものと見なさないようにすることも必要です（以前の ARM アーキテクチャのドキュメントでは、このような値は予測不能と表記されています。これは、アーキテクチャ全体の状態が同じように未指定状態になることを示す UNPREDICTABLE と関連していますが、同じではありません）。

式のサブセットは代入可能です。このため、これらは代入式の左側に置くことができます。このサブセットは、次のもので構成されます。

- 変数
- いくつかの演算子を他の式に適用した結果。代入可能な式を生成可能な言語定義の演算子の記述には、どのような条件で生成が行われるかが指定されています（例えば、演算子が、それ自身が代入可能な式に対して演算を行う、1 つ以上の式が条件に含まれる場合があります）。
- 配列様関数を他の式に適用した結果。配列様関数の記述には、代入可能な式を生成可能な条件が指定されています。

すべての式にはデータ型があります。式の型は、次のように決定されます。

- 定数の場合は、定数の構文
- 変数の場合は、次の 3 つが型のソースとなる可能性があります。
  - — オプションの、前に記述されているデータ型名

— この規則の再帰的な適用により、擬似コードで前もって与えられるデータ型

— 代入により与えられるデータ型（直接代入、またはその変数がメンバであるリストへの代入のいずれかによる）

ある変数について、これら 3 つのデータ型のソースがいずれも存在しない場合、または複数のデータ型のソースが存在し、それらの型が一致しない場合、擬似コードはエラーとなります。

- 言語定義演算子の場合は、演算子の定義
- 関数の場合は、関数の定義

D.4.2 演算子と関数 - 多相性とプロトタイプ

擬似コードの演算子と関数は多相型にすることが可能で、適用されるデータ型によって異なる機能を持つようにできます。結果として生成される演算子または関数のそれぞれの形式は、異なるプロトタイプ定義を持っています。例えば、演算子 + には、整数、実数、ビットストリングスの各種の組み合わせについて機能する形式があります。

多相性のなかでも特によく使われる形式は、長さが異なる複数のビットストリングに対してのものです。これは、プロトタイプ定義で bits(N) や bits(M) などを使うことで表されます。

D.4.3 優先順位の規則

式の優先順位は次の規則で決定されます。

1. 定数、変数、関数呼び出しは、それらの結果を使用しているどの演算子よりも高い優先順位で評価されます。
2. 整数の式は、演算子の通常の規則に従い、*累乗*、*乗算 / 除算*、*加算 / 減算*の順に計算されます。また、連続する乗算 / 除算または加算 / 減算は、左から右の順序で計算されます。
3. その他の式で、順番があいまいになる場合は、優先される演算子を示すために括弧でくくる必要があります。ただし、型の規則に従って使用可能な優先順序がすべて、必然的に同じ結果を導く場合、その必要はありません。例えば、i、j、k が整数の変数の場合、i > 0 && j > 0 && k > 0 は許容されますが、i > 0 && j > 0 || k > 0 は許容されません。

D.5 演算子とエンベデッド関数

このセクションでは、次のトピックについて説明します。

- *すべての型に対して使用できる演算子*
- *ブールの演算*
- *ビットストリング操作*
- *算術演算* : P.AppxD-14

D.5.1 すべての型に対して使用できる演算子

以下の演算子は、すべての型に対して定義されています。

等価と非等価のテスト

同じ型の 2 つの値 x および y が等価であることを式 x == y を使って、等価でないことを式 x != y を使ってテストできます。いずれの場合も、結果は boolean 型です。

'x' ビット、'0' ビット、'1' ビットを含むビットストリング定数を使った、特別な形式の比較を使用できます。'x' ビットと対応するビットは、比較の結果を決定するときに無視されます。例えば、オペコードが 4 ビットのビットストリングの場合、opcode == '1x0x' は opcode<3> == '1' && opcode<1> == '0' と等価です。この特殊な形式は、case ... of ... 構造の when 部分の場合の暗黙的な等価比較でも許可されています。

条件付き選択

x と y が同じ型の 2 つの値で、t が boolean 型の値の場合、if t then x else y は、x および y と同じ型の式で、t が TRUE の場合に x、t が FALSE の場合 y を導出します。

D.5.2 ブールの演算

x が boolean の場合、!x はその論理逆数です。

x と y が boolean の場合、x && y はそれらの AND の結果になります。C 言語の場合と同様に、x が FALSE の場合、結果は FALSE であると y を計算することなく判定されます。

x と y が boolean の場合、x || y はそれらの OR の結果になります。C 言語の場合と同様に、x が TRUE の場合、結果は TRUE であると y を計算することなく判定されます。

x と y が boolean の場合、x ^ y はそれらの排他的論理和の結果になります。

D.5.3 ビットストリング操作

次のビットストリング操作関数が定義されています。

ビットストリング長および最上位ビット

x がビットストリングの場合、ビットストリング長関数 Len(x) はその長さを整数で返し、TopBit(x) はビットストリング抽出を使って x の最も左のビット、つまり x (= x<Len(x) -1> を返します。

ビットストリングの連結と複製

x と y がそれぞれ長さ N および M のビットストリングの場合、x:y は、x と y を左から右への順序で連結させて算出した長さ N+M のビットストリングです。

x がビットストリングで n が n > 0 の整数の場合、Replicate(x,n) は、結合された x の n コピー分の長さ n*Len(x) のビットストリングで、次のようになります。

- Zeros(n) = Replicate('0',n)
- Ones(n) = Replicate('1',n)

ビットストリング抽出

ビットストリング抽出演算子は、ビットストリングを別のビットストリングまたは整数から抽出します。構文は x<integer_list> です。x が抽出される整数またはビットストリングで、<integer_list> は通常の括弧の代わりに不等号で囲まれた整数のリストです。抽出されたビットストリングの長さは、<integer_list> に含まれている整数の数と同じです。

x<integer_list> で、<integer_list> に含まれるそれぞれの整数は次の条件を満たす必要があります。

- >= 0
- x がビットストリングの場合、< Len(x)

x<integer_list> の定義は、integer_list に複数の整数が含まれるかどうかにより異なります。含まれる場合は、x<i,j,k,...,n> は次のような連結として定義されます。

x<i> : x<j> : x<k> : ... : x<n>

integer_list が 1 つの整数 i のみで構成される場合、x<i> は次のように定義されます。

- x がビットストリングの場合、x のビット i が 0 であれば '0'、x のビット i が 1 の場合、'1'。
- x が整数の場合は、y は 0 ～ 2^(i+1) -1 の範囲内で、x modulo 2^(i+1) と一致する一意の整数です。**x<i>** は、y < 2^i の場合は '0'、y >= 2^i の場合は '1' です。

この 2 つ目の定義は、ある整数を、十分な長さを持ったビットストリングとしてのそれの 2 の補数表現と同等のものとして扱います。

<integer_list> で、i >= j の場合、表記 i:j は、i から j までの降順の整数（i および j 自身も含みます）の省略表記となります。例えば、instr<31:28> は instr<31,30,29,28> の省略表記です。

式 x<integer_list> は、x が代入可能なビットストリングで、整数が <integer_list> に複数回現れない場合は、代入可能です。特に、x が割り当て可能なビットストリングで 0 <= i < Len(x) の場合、x<i> は割り当て可能です。

レジスタのエンコード図には、名前付きのビットやマルチビットフィールドが多く示されています。例えば、APSR のエンコード図は、ビット <31> を N として示しています。そのような場合、構文 APSR.N は APSR<31> のより読みやすい同義語として使用されます。

ビットストリングにおける論理演算子

x がビットストリングの場合、NOT(x) は x のビットをすべて論理的に反転した同じ長さのビットストリングです。

x と y が同じ長さのビットストリングの場合、x AND y、x OR y、x EOR y は、x と y の対応するビットに論理積、論理和、排他的論理和を適用した、同じ長さのビットストリングになります。

ビットストリングのカウント

x がビットストリングの場合、BitCount(x) は x に含まれている 1 のビットの数に等しい整数の結果を生成します。

ビットストリングがすべて 0 またはすべて 1 であるかどうかのテスト

x がビットストリングの場合、IsZero(x) は、x のすべてのビットが 0 のときに TRUE を、1 つでも 1 がある場合は FALSE を返します。IsZeroBit(x) は、x のビットがすべて 0 のときは '1' を、1 つでも 1 がある場合は '0' を生成します。

IsOnes(x) および IsOnesBit(x) も同様の機能を持ち、次のような結果となります。

```
IsZero(x) = (BitCount(x) == 0)

IsOnes(x) = (BitCount(x) == Len(x) )

IsZeroBit(x) = if IsZero(x) then '1' else '0'

IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

ビットストリングでセットされている最下位および最上位ビット

x がビットストリングで N = Len(x) の場合、次のように関数が定義されます。

- LowestSetBit(x) は、含まれる 1 のビットの中で最小のビット番号です。すべてのビットが 0 の場合、LowestSetBit(x) = N です。
- HighestSetBit(x) は、含まれる 1 のビットの中で最大のビット番号です。すべてのビットが 0 の場合、HighestSetBit(x) = -1 です。
- CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x) では、範囲は 0 ～ N で、x の左端から数えた 0 のビットの数です。
- CountLeadingSignBits(x) = CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>) では、範囲は 0 ～ N-1 で、x の左端から数えた、符号ビットと同じビットがコピーされている数（符号ビット自体を除く）です。

ビットストリングのゼロ拡張および符号拡張

x がビットストリングで i が整数の場合、ZeroExtend(x,i) は、必要な個数だけ 0 のビットを左側に加えて i ビットの長さに拡張された x です。つまり、i == Len(x) の場合、ZeroExtend(x,i) = x です。i > Len(x) の場合は次のように定義されます。

```
ZeroExtend(x,i) = Zeros(i-Len(x) ) : x
```

x がビットストリングで i が整数の場合、SignExtend(x,i) は、必要な個数だけ最も左のビットのコピーをその左側に加えて i ビットの長さに拡張された x です。つまり、i == Len(x) の場合、SignExtend(x,i) = x です。i > Len(x) の場合は次のように定義されます。

```
SignExtend(x,i) = Replicate(TopBit(x) , i-Len(x) ) : x
```

i < Len(x) の可能性がある場合に ZeroExtend(x,i) または SignExtend(x,i) を使用すると、擬似コードエラーとなります。

ビットストリングの整数への変換

x がビットストリングの場合、SInt(x) は 2 の補数表現が x となる整数です。

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

UInt(x) は、符号なし表現が x となる整数です。

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

Int(x, unsigned) は、2 つ目の引数の値に応じて、SInt(x) または UInt(x) を返します。

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

D.5.4 算術演算

ほとんどの擬似コード算術演算は整数または実数値に対して実行され、オペランドはビットストリングからの変換により取得され、結果は再度ビットストリングに戻されます。これらのデータ型は制限の無い数学的な型で、オーバフローや類似のエラーに関する問題は発生しません。

単項のプラス、マイナス、絶対値

x が整数または実数の場合、+x は変更されていない x、-x は符号が反転された x、ABS(x) は x の絶対値です。これらの 3 つはすべて、x と同じ型です。

加算と減算

x と y が整数または実数の場合、x+y と x-y がそれらの合計および差異です。x と y が両方とも integer 型の場合は、結果も両方とも integer 型です。それ以外の場合、結果は real 型です。

加算と減算は、擬似コードで特によく行われる算術演算です。このため、加算と減算の動作の定義をビットストリングの演算にも直接適用するのが便利です。

x と y が同じ長さのビットストリング、つまり N = Len(x) = Len(y) の場合、x+y と x-y は、それらを整数に変換して加算または減算を行った結果の最下位の N ビットです。符号付きと符号なし変換は、いずれも同じ結果となります。

```
x+y = (SInt(x) + SInt(y) ) <N-1:0>
    = (UInt(x) + UInt(y) ) <N-1:0>

x-y = (SInt(x) - SInt(y) ) <N-1:0>
    = (UInt(x) - UInt(y) ) <N-1:0>
```

x が長さ N のビットストリングで、y が整数の場合、x+y と x-y は長さ N のビットストリングで、それぞれ x+y = x + y<N-1:0> および x-y = x - y<N-1:0> と定義されます。同様に、x が整数で y が長さ M のビットストリングの場合、x+y と x-y は長さ M のビットストリングで、それぞれ x+y = x<M-1:0> + y および x-y = x<M-1:0> - y と定義されます。

比較

x と y が整数または実数の場合、x == y、x != y、x < y、x <= y, x > y、x >= y は、それらの間での等価、不等価、未満、以下、より大きい、以上の比較になり、結果はブール型です。== および != の場合、同じ型の 2 つの値に適用される一般的な定義が拡張され、整数と実数の間にも適用されます。

操作

x と y が整数または実数の場合、x * y は x と y の積です。x と y の両方が integer 型の場合は結果も integer 型で、それ以外の場合、結果は real 型です。

除算とモジュロ

x と y が整数または実数の場合、x / y は x を y で割った結果で、常に実数型です。

x と y が整数の場合、x DIV y と x MOD y は次のように定義されます。

```
x DIV y = RoundDown(x / y)
x MOD y = x - y * (x DIV y)
```

y が 0 である可能性がある場合に、x / y、x MOD y、x DIV y を使用すると擬似コードエラーとなります。

平方根

x が整数または実数の場合、Sqrt(x) は平方根で、常に実数型です。

丸めとアライン

x が実数の場合、次のように定義されます。

- RoundDown(x) は、n <= x である最大の整数 n を生成します。

- RoundUp(x) は、n >= x である最小の整数 n を生成します。
- RoundTowardsZero(x) は、x > 0.0 の場合は RoundDown() 、x == 0.0 の場合は 0、x < 0.0 の場合は RoundDown(x) を生成します。

x と y が整数の場合、Align(x,y) = y * (x DIV y) は整数型です。

x がビットストリングで y が整数の場合、Align(x,y) = (Align(UInt(x) ,y)) <Len(x) -1:0> は x と同じ長さのビットストリングです。

y が 0 の可能性がある場合、Align(x,y) はどの形式でも擬似コードエラーとなります。実際には、Align(x,y) は y が 2 の定数乗である場合のみ使用され、y = 2^n とともに使用されるビットストリング形式には、下位ビット n を強制的に 0 にして引数を生成する効果があります。

スケーリング

n が整数の場合、2^n は 2 を n 乗した結果になり、real 型です。

x と n が整数の場合、次のように定義されます。

- x << n = RoundDown(x * 2^n)
- x >> n = RoundDown(x * 2^(-n))

最大値と最小値

x と y が整数または実数の場合、Max(x,y) と Min(x,y) が最大値および最小値です。x と y が両方とも integer 型の場合は、結果も両方とも integer 型です。それ以外の場合、結果は real 型です。

D.6 文とプログラム構造

このセクションでは、擬似コードで使用される制御文について説明します。

D.6.1 単純な文

以下に示す単純な文は、すべてセミコロンで終わる必要があります。

代入

代入文は次の形式です。

```
<assignable_expression> = <expression>;
```

プロシージャ呼び出し

プロシージャ呼び出しは次の形式です。

```
<procedure_name>(<arguments>) ;
```

リターン文

プロシージャからの復帰は次の形式です。

```
return;
```

関数の戻り値は次の形式です。

```
return <expression>;
```

ここで、<expression> は関数プロトタイプ行で宣言された型です。

UNDEFINED

この文は次の形式です。

```
UNDEFINED;
```

現在の擬似コードにより定義された動作を置き換える特殊なケースのひとつであることを示します（特殊なケースが適用されることを判定するために必要な動作を除きます）。置き換え動作では、未定義命令例外が取得されます。

UNPREDICTABLE

この文は次の形式です。

```
UNPREDICTABLE;
```

現在の擬似コードにより定義された動作を置き換える特殊なケースのひとつであることを示します（特殊なケースが適用されることを判定するために必要な動作を除きます）。置き換え動作は、アーキテクチャ上では定義されていません。ソフトウェアは、この動作に依存しないようにする必要があります。置き換え動作が、セキュリティホールや、システムの停止またはハングの原因にならないようにする必要があります。また、この動作がソフトウェアに有用な情報を提供するものとは示さないことが必要です。

SEE...

この文は次の形式です。

SEE <reference>;

現在の擬似コードにより定義された動作を置き換える特殊なケースのひとつであることを示します（特殊なケースが適用されることを判定するために必要な動作を除きます）。置き換え動作では、現在の擬似コードの結果としては何も起きません。必要な動作は別の擬似コードで定義されています。<reference> は、擬似コードのその部分がどこにあるのかを示しています。

IMPLEMENTATION_DEFINED

この文は次の形式です。

IMPLEMENTATION_DEFINED <text>;

動作が実装定義となる特殊なケースのひとつであることを示します。詳細は、文の後に続く text に記載されています。

SUBARCHITECTURE_DEFINED

この文は次の形式です。

```
SUBARCHITECTURE_DEFINED <text>;
```

動作がサブアーキテクチャ定義となる特殊なケースのひとつであることを示します。詳細は、文の後に続く text に記載されています。

D.6.2 複合文

インデントは通常、複合文の構造を示すために使用されます。if ... then ... else ... などの構造に含まれる文、またはプロシージャや関数の定義は、文自体よりもより深くインデントされていて、それらの終わりは元のインデントレベルまたはそれ以下のレベルに戻ることで示されます。

インデントは通常、レベルごとに 4 つのスペースにより示されます。

if ... then ... else ...

複数行の if ... then ... else ... 構造は、次の形式です。

```
if <boolean_expression> then
    <statement 1>
    <statement 2>
    ...
```

```
    <statement n>
elsif <boolean_expression> then
    <statement a>
    <statement b>
    ...
    <statement z>
else
    <statement A>
    <statement B>
    ...
    <statement Z>
```

else およびそれに続く文はオプションです。

```
if <boolean_expression> then
    <statement 1>
    <statement 2>
    ...
    <statement n>
elsif <boolean_expression> then
    <statement a>
    <statement b>
    ...
    <statement z>
else
    <statement A>
    <statement B>
    ...
    <statement Z>
```

elsif と、その後のインデントされた文で構成されるブロックはオプションです。また、そのようなブロックを複数使用することができます。

else と、その後のインデントされた文で構成されるブロックはオプションです。

then および else（存在する場合）部分に単純な文しかない場合は、次のように省略された 1 行の形式を使用することができます。

```
if <boolean_expression> then <statement 1>

if <boolean_expression> then <statement 1> else <statement A>

if <boolean_expression> then <statement 1> <statement 2> else <statement A>
```

注

これらの形式では、<statement 1>、<statement 2>、および <statement A> は、セミコロンで終了する必要があります。この点と、else 部分がオプションであることが、if ... then ... else ... 式との相違点です。

repeat ... until ...

A `repeat ... until ...`ç\ë¢ÇÕÅAéüÇÃå`éÆÇÝÇ²ÅB


```
repeat
    <statement 1>
    <statement 2>
    ...
    <statement n>
until <boolean_expression>;
```

while ... do

A while ... do 構造は、次の形式です。

```
while <boolean_expression>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

for ...

A for ... 構造は、次の形式です。

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

case ... of ...

A case ... of ... 構造は、次の形式です。

```
case <expression> of
    when <constant values>
        <statement 1>
        <statement 2>
        ...
        <statement n>
    ... more "when" groups ...
    otherwise
        <statement A>
        <statement B>
        ...
        <statement Z>
```

ここで、<constant values> は、<expression> と同じ型の 1 つ以上の定数値を、カンマで区切って並べたものです。when と otherwise 部分が単純な文のみを含む場合は、それらの部分が短縮された 1 行の形式を使用することができます。

<expression> がビットストリング型の場合、<constant values> には 'x' ビットを含むビットストリング定数も含めることができます。詳細については、P.AppxD-11「*等価と非等価のテスト*」を参照して下さい。

プロシージャと関数の定義

プロシージャの定義は次の形式です。

```
<procedure name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

ここで、<argument prototypes> はカンマで区切られた 0 個以上の引数の定義です。それぞれの引数の定義は、型名と、それに続く引数名で構成されます。

注

最初のプロトタイプ行はセミコロンで終わりません。これにより、プロシージャ呼び出しと区別することができます。

関数の定義も同様ですが、関数の戻り値の型も宣言されます。

```
<return type> <function name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

配列様の関数も同様の形式ですが、この場合は角括弧が使用されます。

```
<return type> <function name>[<argument prototypes>]
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

また、配列様の関数には、一般に代入プロトタイプも含まれます。

```
<function name>[<argument prototypes>] = <value prototypes>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

D.6.3 コメント

擬似コードのコメントには 2 つの形式があります。

- // で始まるコメントは、行の終わりで終了します。
- /* で始まるコメントは、*/ で終了します。

D.7 その他のヘルパプロシージャおよび関数

このセクションで説明されている関数は擬似コード仕様の一部ではありません。これらは、アーキテクチャ固有のジョブを実行するために擬似コードにより使用されるヘルパプロシージャおよび関数です。それぞれのプロシージャと関数について、簡単な記述と擬似コードプロトタイプを示します。いくつかについては、擬似コードの定義も示されています。

D.7.1 ALUWritePC()

このプロシージャは、ADD (register) や MOV (register) データ処理命令による PC への書き込みに対して、意味的に正しく PC に値を書き込みます。

```
ALUWritePC(bits(32) value)
```

D.7.2 ArchVersion()

この関数は、アーキテクチャのメジャーバージョン番号を返します。

```
integer ArchVersion()
```

D.7.3 BadReg()

この関数は、多くの Thumb レジスタ指定子で許可されていないレジスタ番号 13 と 15 に対するをチェックします。

```
boolean BadReg(integer n)

return n == 13 || n == 15;
```

D.7.4 BigEndian()

ロード / ストア操作が現在ビッグエンディアンの場合、この関数は TRUE を返します。リトルエンディアンの場合は FALSE を返します。

```
boolean BigEndian()
```

D.7.5 BigEndianReverse()

この関数は、ビッグエンディアンのアクセスが要求される場合に、値のバイトを反転するために使用されます。

D.7.6 BranchWritePC()

このプロシージャは、単純な分岐による PC への書き込みに対して、意味的に正しく PC に値を書き込みますーすなわちどのような状況においても単なる PC の変更です。

```
BranchWritePC(bits(32) value)
```

D.7.7 BreakPoint()

このプロシージャは、デバッグブレークポイントを引き起こします。

D.7.8 BXWritePC()

このプロシージャは、インターワーキング命令による PC への書き込みに対して、意味的に正しく PC に値を書き込みます。これは、すべての状況で BX と同様のインターワーキング動作を行うことを意味します。

```
BXWritePC(bits(32) value)
```

注

M プロファイルは Thumb 実行状態のみをサポートします。命令実行状態を変更しようとすると、例外が発生します。

D.7.9 CallSupervisor()

M プロファイルでは、このプロシージャは SVCall 例外を引き起こします。

D.7.10 CheckPermissions()

ValidateAddress() とともに使用して、メモリアクセス許可をチェックし、アクセス違反例外があった場合に例外を発生します。

D.7.11 ClearEventRegister()

このプロシージャは、現在のプロセッサのイベントレジスタをクリアします。イベントレジスタの詳細については P.AppxD-25「*EventRegistered()*」を参照して下さい。

D.7.12 ClearExclusiveByAddress()

ローカルモニタを、アドレスタグが一致しているオープンアクセス状態に戻します。関連するグローバルモニタがクリアされている場合、これは実装定義です。

D.7.13 ClearExclusiveMonitors()

このプロシージャは、排他ロード / ストア命令に使用されるローカルモニタをクリアします。関連するグローバルモニタがクリアされている場合、これは実装定義です。

D.7.14 ConditionPassed()

この関数は、次の条件に基づいて命令の条件テストを実行します。

- 2 つの Thumb 条件付き分岐エンコード（B 命令のエンコード T1 および T3）
- 他の Thumb 命令の xPSR.IT[7:0] ビットの現在の値

```
boolean ConditionPassed()
```


D.7.15 Coproc_Accepted()

この関数は、コプロセッサが命令を受け付けるかどうかを判定します。

```
boolean Coproc_Accepted(integer cp_num, bits(32) instr)
```

D.7.16 Coproc_DoneLoading()

この関数は、LDC 命令で十分な数のワードがロードされたかどうかを判定します。

```
boolean Coproc_DoneLoading(integer cp_num, bits(32) instr)
```

D.7.17 Coproc_DoneStoring()

この関数は、STC 命令で十分な数のワードがストアされたかどうかを判定します。

```
boolean Coproc_DoneStoring(integer cp_num, bits(32) instr)
```

D.7.18 Coproc_GetOneWord()

この関数は、MRC 命令のワードをコプロセッサから取得します。

```
bits(32) Coproc_GetOneWord(integer cp_num, bits(32) instr)
```

D.7.19 Coproc_GetTwoWords()

この関数は、MRRC 命令の 2 ワードをコプロセッサから取得します。

```
(bits(32) , bits(32) ) Coproc_GetTwoWords(integer cp_num, bits(32) instr)
```

D.7.20 Coproc_GetWordToStore()

この関数は、STC 命令でストアする次のワードをコプロセッサから取得します。

```
bits(32) Coproc_GetWordToStore(integer cp_num, bits(32) instr)
```

D.7.21 Coproc_InternalOperation()

このプロシージャは、CDP 命令により要求された内部操作を行うようにコプロセッサに指示します。

```
Coproc_InternalOperation(integer cp_num, bits(32) instr)
```

D.7.22 Coproc_SendLoadedWord()

このプロシージャは、LDC 命令のためにロードされたワードをコプロセッサに送ります。

```
Coproc_SendLoadedWord(bits(32) word, integer cp_num, bits(32) instr)
```

D.7.23 Coproc_SendOneWord()

このプロシージャは、MCR 命令のワードをコプロセッサに送ります。

```
Coproc_SendOneWord(bits(32) word, integer cp_num, bits(32) instr)
```

D.7.24 Coproc_SendTwoWords()

このプロシージャは、MCRR 命令の 2 ワードをコプロセッサに送ります。

```
Coproc_SendTwoWords(bits(32) word1, bits(32) word2, integer cp_num,
                    bits(32) instr)
```

D.7.25 DataMemoryBarrier()

このプロシージャは、データ メモリバリアを生成します。

```
DataMemoryBarrier(bits(4) option)
```

D.7.26 DataSynchronizationBarrier()

このプロシージャは、データ同期バリアを生成します。

```
DataSynchronizationBarrier(bits(4) option)
```

D.7.27 Deactivate()

この関数は、PopStack() とともに例外からの復帰に使用されます。

D.7.28 DecodeImmShift() , DecodeRegShift()

これらの関数は、標準的な *2 ビットの型*、*5 ビットの数量*、*2 ビットの型*の形式のデコードを、それぞれイミディエートおよびレジスタシフトについて実行します。詳細については、P.A6-13「*シフト操作*」を参照して下さい。

D.7.29 DefaultAttrs()

システム メモリマップからメモリ属性を決定します。

D.7.30 DefaultDecode()

ValidateAddress() とともに使用して、MPURASR からメモリ属性を決定するのに使用します。

D.7.31 EventRegistered()

現在のプロセッサ上のイベントレジスタがセットされている場合、この関数は TRUE を返します。クリアされている場合は FALSE を返します。イベントレジスタは、次のいずれかのイベントの結果としてセットされます。

- IRQ 割り込み（CPSR の I ビットによってマスクされていない場合）

- FIQ 割り込み（CPSR の F ビットによってマスクされていない場合）
- 不正確データアボート（CPSR の A ビットによってマスクされていない場合）
- デバッグが許可されている場合、デバッグ開始要求
- マルチプロセッサシステムで、プロセッサが Hint_SendEvent() を実行した結果として、そのプロセッサより送信されるイベント
- 例外からの復帰
- 実装固有の理由。これは実装定義の場合もありますが、任意に発生することもあります。

リセット時のイベントレジスタの状態は未知です。

D.7.32 EncodingSpecificOperations()

このプロシージャは、エンコード固有の擬似コードを命令エンコードのために呼び出し、P.AppxD-2「*命令エンコード図および擬似コード*」で説明されているように、そのエンコードの「必須」ビットをチェックします。

D.7.33 ExceptionTaken()

この関数は、PushStack () とともに例外の開始のために使用されます。

D.7.34 ExclusiveMonitorsPass()

この関数は、排他ストア命令が成功したかどうかを判定します。排他モニタをまだ所有している場合、排他ストア命令は成功します。

D.7.35 FindPriv()

特権実行かどうかを判定するために使用されます。

D.7.36 Hint_Debug()

このプロシージャは、デバッグシステムにヒントを提供します。

```
Hint_Debug(bits(4) option)
```

D.7.37 Hint_PreloadData()

このプロシージャは、*データのプリロード*のヒントを実行します。

```
Hint_PreloadData(bits(32) address)
```

D.7.38 Hint_PreloadInstr()

このプロシージャは、*命令のプリロード*のヒントを実行します。

```
Hint_PreloadInstr(bits(32) address)
```

D.7.39 Hint_SendEvent()

このプロシージャは*イベント送信*のヒントを実行します。

D.7.40 Hint_Yield()

このプロシージャは*イールド*のヒントを実行します。

D.7.41 InITBlock()

現在 IT ブロック内部が実行されている場合、この関数は TRUE を返します。それ以外の場合は FALSE を返します。

```
boolean InITBlock()
```

D.7.42 InstructionSynchronizationBarrier()

このプロシージャは、命令同期バリアを生成します。

```
InstructionSynchronizationBarrier(bits(4) option)
```

D.7.43 IntegerZeroDivideTrappingEnabled()

整数除算命令 SDIV および UDIV で、0 による除算のトラップが許可されている場合、この関数は TRUE を返します。それ以外の場合は FALSE を返します。

M プロファイルでは、これはコンフィギュレーション制御レジスタの DIV_0_TRP ビットにより制御されます。このビットが 1 の場合は TRUE が返され、0 の場合は FALSE が返されます。

D.7.44 IsAligned()

メモリアクセスでのアライメントのチェックで使用されます。

D.7.45 LastInITBlock()

現在の命令が IT ブロックの最後の命令である場合、この関数は TRUE を返します。それ以外の場合は FALSE を返します。

D.7.46 LoadWritePC()

このプロシージャは、ロード命令による書き込みの BX と同様のインターワーキング動作で、PC に値を書き込みます。

```
LoadWritePC(bits(32) value)
```


注

M プロファイルは Thumb 実行状態のみをサポートします。命令実行状態を変更しようとすると、例外が発生します。

D.7.47 MarkExclusiveGlobal()

ProcessorID() と対応するグローバルモニタを排他アクセス状態に設定し、タグアドレスを保存します。

D.7.48 MarkExclusiveLocal()

ProcessorID() と対応するローカルモニタを排他アクセス状態に設定します。アドレスタグが保存されるかどうかは実装定義です。

D.7.49 Mem*[]

次に示すすべてのメモリアクセス ヘルパ関数に、後の注が適用されます。

- MemA[]
- MemAA[]
- MemA_unpriv[]
- MemU[]
- MemU_unpriv[]

注

ARM アーキテクチャでは、メモリへのインタフェースの特性や、このインタフェース上のメモリトランザクションの信号は規定されていません。インタフェースの詳細は、第 A3 章 *「ARM アーキテクチャのメモリモデル」* で説明されている ARM アーキテクチャの制約の範囲内で実装定義です。

D.7.50 MemA[]

この配列様関数は、アラインされていることが要求されるメモリアクセスを、現在の特権レベルを使用して実行します。

D.7.51 MemAA[]

この配列様関数は、アラインされていることとアトミックであることが要求されるメモリアクセスを、現在の特権レベルを使用して実行します。

D.7.52 MemA_unpriv[]

この配列様関数は、アラインされていることが要求されるメモリアクセスを、現在の特権レベルにかかわらず非特権アクセスとして実行します。

D.7.53 MemU[]

この配列様関数は、アンアラインドであることが許可されているメモリアクセスを、現在の特権レベルを使用して実行します。

D.7.54 MemU_unpriv[]

この配列様関数は、アンアラインドであることが許可されているメモリアクセスを、現在の特権レベルにかかわらず非特権アクセスとして実行します。

D.7.55 PopStack()

この関数は、例外からの復帰時にスタックからコンテキストを回復するために使用されます。

D.7.56 ProcessorID()

実行中のプロセッサを識別します。

D.7.57 PushStack()

この関数は、例外の開始時にコンテキストをスタックにストアするために使用されます。

D.7.58 R[]

この配列様関数は、レジスタの読み出しまたは書き込みを行います。レジスタ 13、14、または 15 の読み出しは、それぞれ SP、LR、PC を読み出します。また、レジスタ 13 または 14 への書き込みは、それぞれ SP または LR へ書き込みます。

```
bits(32) R[integer n]

R[integer n] = bits(32) value;
```

D.7.59 RaiseCoprocessorException()

このプロシージャは、拒否されたコプロセッサ命令に対して UsageFault 例外を発生させます。

D.7.60 RaiseIntegerZeroDivide()

このプロシージャは、整数除算命令 SDIV および UDIV での 0 による除算に対して、適切な例外を発生させます。

M プロファイルの場合、これは UsageFault 例外です。

D.7.61 SetExclusiveMonitors()

このプロシージャは、排他ロード命令のために排他モニタを設定します。

D.7.62 SetPending()

このプロシージャは、対応する例外の状態を保留に設定します。他の例外状態の定義については、P.B1-4「*例外*」を参照して下さい。

D.7.63 Shift() , Shift_C()

これらの関数は、標準的な ARM シフトを値に対して実行し、結果の値を返します。Shift_C() の場合は、キャリーアウト ビットも返されます。詳細については、P.A6-13「*シフト操作*」を参照して下さい。

D.7.64 StartITBlock()

このプロシージャは、指定された*最初の条件*と *マスク*の値で IT ブロックを開始します。

```
StartITBlock(bits(4) firstcond, bits(4) mask)
```

D.7.65 ThisInstr()

この関数は、現在実行されている命令を返します。現在のところ、これは 32 ビット命令エンコードでのみ使用されます。

```
bits(32) ThisInstr()
```

D.7.66 ThumbExpandImm() , ThumbExpandImmWithC()

これらの関数は、Thumb データ処理のイミディエートを示す 12 ビットの値を、標準の方法で 32 ビット値に拡張します。WithC 関数はキャリーアウト ビットも生成します。詳細については、P.A5-15「*Thumb-2 命令での修飾イミディエート定数*」を参照して下さい。

D.7.67 ValidateAddress()

メモリアクセスの一部として与えられたアドレス対応したアクセス許可およびメモリ属性を決定し、必要に応じてアクセス違反例外を発生させます。

D.7.68 WaitForEvent()

このプロシージャは、マルチプロセッサシステム中の任意のプロセッサが SEV 命令を実行するか、次のいずれかの条件がプロセッサ自体で発生するまで、そのプロセッサの実行を中断します。

- SEV（イベント送信）命令を使用して、他のプロセッサからイベントが送信される。
- システム制御レジスタの SEVONPEND が設定されている場合に、例外が保留状態に移行する。
- 現在アクティブな例外のすべてを横取りする優先度の非同期例外が発生する。

- デバッグが許可された状態で、デバッグイベントが発生する。
- 実装固有の理由。これは実装定義の場合もありますが、任意に発生することもあります。
- リセット

イベントレジスタがセットされている場合、イベント待ち命令はイベントレジスタをクリアして直ちに復帰します。

中断期間が過ぎた後での処理の再開が ClearEventRegister() を起こすかどうかは実装定義です。

D.7.69 WaitForInterrupt()

このプロシージャは、次のいずれかの条件がプロセッサに発生するまで、プロセッサの実行を中断します。

- 現在アクティブな例外のすべてを横取りする優先度の非同期例外が発生する。
- デバッグが許可された状態で、デバッグイベントが発生する。
- 実装固有の理由。これは実装定義の場合もありますが、任意に発生することもあります。
- リセット

用語集

AAPCS
ARM アーキテクチャのプロシージャ呼び出し標準。

アドレシングモード
ロード / ストア命令で使用されるメモリアドレスを計算する方法。

アラインド
データ項目が、そのデータサイズを格納できる最小の 2 のべき乗の値で割り切れるようなアドレスに格納されている場合、そのデータ項目はアラインしていると呼びます。したがって、アラインしているハーフワード、ワード、ダブルワードのアドレスは、それぞれ 2、4、8 で割り切れます。
アラインドアクセスは、それがアクセスする要素のサイズにアドレスがアラインしているようなアクセスを指します。

APSR アプリケーションプログラム ステータスレジスタ参照。

アプリケーションプログラム ステータスレジスタ
命令の結果に関するステータス情報を示すビットである、xPSR の N、Z、C、V ビットを含むレジスタ。詳細については、P.B1-7「*専用プログラム ステータスレジスタ (xPSR)*」を参照して下さい。

アトミック性
シングルコピーのアトミック性、またはマルチコピーのアトミック性を意味する用語。ARM アーキテクチャで使用されるアトミック性の形式は、P.A3-21「*ARM アーキテクチャにおけるアトミック性*」に定義されています。
マルチコピーのアトミック性、シングルコピーのアトミック性も参照。

バンクレジスタ
複数のインスタンスを持つレジスタで、使用されるインスタンスはプロセッサのモード、セキュリティ状態、その他プロセッサの状態に応じて変化します。

ベースレジスタ
命令のアドレス計算の基準値として、ロード / ストア命令で指定されるレジスタ。命令とそのアドレシングモードによっては、メモリに送られる仮想アドレスを形成するために、ベースアドレスの値にオフセットを加算または減算することができます。

ベースレジスタ ライトバック
アドレス計算時に、修飾された値をベースレジスタへ書き戻すことを指します。

ビッグエンディアン メモリ
次の条件を満たすメモリ。

- ワードアラインされたアドレスのバイトまたはハーフワードが、そのアドレスのワード内で最上位のバイトまたはハーフワードである。
- ハーフワードアラインされたアドレスのバイトが、そのアドレスのハーフワード内で最上位バイトである。

ブロッキング
操作が完了するまで、以後の命令の実行を許可しないような操作を指します。

非ブロッキングの操作では、操作が完了する前に次の命令を実行することが許可され、例外が発生した場合にはコアに例外が通知されません。これにより、正確なプロセッサ状態を保持しなくても、非ブロッキング操作を実行している間に後続の命令を完了するような実装が可能です。

分岐予測
プロセッサが、その先プリフェッチする実行パスを選択すること（プリフェッチ参照）。例えば、分岐命令の後でプロセッサは分岐に続く命令かまたは分岐先の命令のどちらかをプリフェッチできます。

ブレークポイント
特定の命令の実行によりトリガされるデバッグイベントで、命令のアドレス、命令が実行されたときのプロセッサの状態、またはその両方によって指定されます。

バイト　8 ビットのデータアイテム。

キャッシュ
高速なメモリのブロックで、プロセッサがアクセスするメモリ位置に応じて自動的にアドレスが変更されます。キャッシュはメモリアクセスの平均速度を向上することを目的としています。

キャッシュの競合
ある特定のキャッシュライン中で利用しているキャッシュラインの数がキャッシュのセットアソシエティビティを超えたとき。この場合は、メインメモリの動作が増大して、性能が低下します。

キャッシュヒット
アクセスされるデータがすでにキャッシュにあるため、高速で処理できるメモリアクセス。

キャッシュライン
キャッシュ内の記憶域の基本単位。このサイズは常に 2 のべき乗（通常は 4 または 8 ワード）で、適切なメモリ境界にアラインしている必要があります。*メモリ キャッシュラインは*、キャッシュラインと同じサイズ、同じアライメントのメモリブロックです。メモリ キャッシュラインを大まかに、キャッシュラインとだけ呼ぶこともあります。

キャッシュミス
アクセスされるデータがキャッシュに存在しないため、高速で処理できないメモリアクセス。

呼び出し先保存レジスタ
呼び出されたプロシージャが保存する必要のあるレジスタ。呼び出し先保存レジスタを保存するため、呼び出されたプロシージャは一般にそのレジスタを一切使用しないか、プロシージャの開始時にレジスタをスタックに保存して、プロシージャの終了時に再ロードします。

呼び出し元保存レジスタ
呼び出されたプロシージャが保存する必要のないレジスタ。呼び出し元のプロシージャが値を保持する必要がある場合、呼び出し元で保存と再ロードを行う必要があります。

クリア　レジスタまたはレジスタのフィールドに関連した用語。ビットの値が 0（またはビットフィールドがすべて 0）である、または 0 やすべて 0 の値を書き込むことを意味します。

条件付き実行
条件コードフラグが、命令の実行開始時に該当する条件が真であることを示している場合、命令が通常に実行されることを意味します。それ以外の場合、命令は何も実行しません。

コンフィギュレーション
リセット時またはリセットの直後に行われる設定で、通常はプログラムの実行を通して静的に保持されます。

コンテキストスイッチ
スレッドやプロセス間で切り替えを行うとき、計算の状態を保存および復元すること。本書では、コンテキストスイッチという用語はオペレーティングシステムによってコンテキストが切り替えられるすべての状況を指し、アドレス空間が変更されるかどうかは関係しません。

DCB　デバッグ制御ブロックの略。システム制御空間（SCS 参照）内で、デバッグ機能のレジスタサポートに特別に割り当てられた領域。

デジタル信号処理 (DSP)
サンプリングされてデジタル形式に変換された信号を処理するために使用される各種のアルゴリズムを指します。このようなアルゴリズムでは、飽和演算が多く使用されます。

ダイレクト メモリアクセス
プロセッサによるデータへのアクセスを行わず、メインメモリへ直接アクセスする操作。

変更不可フィールド (DNM)
ソフトウェアで値を変更できないことを意味します。DNM フィールドを読み出した場合の値は未知で、同じプロセッサの同じフィールドから読み出した値のみを書き込むことができます。

ダブルワード
64 ビットのデータアイテム。ARM システムでのダブルワードは、通常は最低でもワードアラインドです。

ダブルワードアラインド
アドレスが 8 の倍数であることを意味します。

DSP　デジタル信号処理参照。

DWT　データウォッチポイントおよびトレースの略。ARM デバッグアーキテクチャの一部です。

エンディアン形式
システムのメモリマッピングの要素。ビッグエンディアンおよびリトルエンディアン参照。

EPSR 実行プログラム ステータスレジスタ参照。

ETM エンベデッドトレースマクロセルの略。ARM デバッグアーキテクチャの一部です。

例外
イベントを処理します。例えば、外部割り込みや未定義命令を処理できます。

例外ベクタ
下位メモリにある多くの固定アドレスの 1 つ。ハイベクタが構成されている場合は上位メモリにあります。

実行プログラムステータス レジスタ
実行状態ビットを含むレジスタで、xPSR の一部。詳細については、P.B1-7「*専用プログラム ステータスレジスタ (xPSR)*」を参照して下さい。

実行ストリーム
プログラムのシーケンシャルな実行により実行される命令のストリーム。

明示的アクセス
CPU で実行されるロード / ストア命令により生成されるメモリからの読み出し、またはメモリへの書き込み。L1 DMA アクセスや、ハードウェア変換テーブルアクセスにより生成される読み出しや書き込みは、明示的アクセスではありません。

フォルト
なんらかの形態のシステムエラーによる例外。

汎用レジスタ
32 ビット汎用整数レジスタ R0 ～ R15 のうち 1 つ。R15 はプログラムカウンタを保持しているため、R0 ～ R14 には適用されない多くの使用制限があることに注意して下さい。

ハーフワード
16 ビットのデータアイテム。ARM システムでのハーフワードは、通常はハーフワードアラインドです。

ハーフワードアラインド
アドレスが 2 の倍数であることを意味します。

上位レジスタ
ARM コアレジスタの 8 ～ 15 で、一部の Thumb 命令によりアクセス可能です。

イミディエートおよびオフセットフィールド
これらのフィールドは、特記されていない限り符号なしです。

イミディエート値
命令に直接エンコードされ、命令の実行時に数値データとして使用される値。多くの ARM および Thumb 命令では、小さな数値をイミディエート値として、その数値を使用する命令にエンコードできます。

IMP 該当するビットの動作が実装定義であることを示すため、図中で使用される略号。

実装定義
動作がアーキテクチャ上で定義されていないために、実装ごとに定義して文書化する必要があることを意味します。

インデクスレジスタ
一部のロード / ストア命令で指定されるレジスタ。このレジスタの値は、メモリに送出するアドレスを生成するために、オフセットとしてベースレジスタの値に加算または減算されます。一部のアドレシングモードでは、必要に応じて、インデクスレジスタの値を加算または減算する前にシフトできます。

インラインリテラル
コード自体と同じ領域に保持される定数アドレス、およびその他のデータアイテム。これらはコンパイラにより自動的に生成され、アセンブラのコードにも含まれていることがあります。

割り込みプログラムステータス レジスタ
アプリケーションスレッドや例外ハンドラが現在プロセッサで実行されているかどうかのステータス情報を示すレジスタ。例外ハンドラが実行されている場合、レジスタは例外のタイプの情報を示しています。このレジスタは、xPSR の一部です。詳細については、P.B1-7「*専用プログラム ステータスレジスタ (xPSR)*」を参照して下さい。

インターワーキング
ARM と Thumb 両方の実行状態をサポートするアーキテクチャバリアントで、ARM と Thumb のコードの間で分岐を行う方法。

IPSR　割り込みプログラムステータス レジスタ参照。

IT ブロック
If-Then(IT) 命令に続く最大 4 つの命令のブロック。ブロック内の各命令は条件付きです。命令の条件はすべて同じ、または一部の命令が他の命令に使用される条件の逆の条件を使用します。詳細については、P.A6-78「*IT*」を参照して下さい。

ITM　計装トレースマクロセルの略。ARM デバッグアーキテクチャの一部です。

リトルエンディアンメモリ
次の条件を満たすメモリ。

- ワードアラインされたアドレスのバイトまたはハーフワードが、そのアドレスにあるワード内の最下位バイトまたはハーフワードである。
- ハーフワードアラインされたアドレスのバイトが、そのアドレスのハーフワード内の最下位バイトである。

ロード / ストアアーキテクチャ
データ処理動作が、直接メモリの内容に対して行われるのではなく、レジスタの内容に対してのみ行われるアーキテクチャ。

ロング分岐
ロード命令を使用して、4GB のアドレス空間内で任意の場所へ分岐すること。

メモリバリア
詳細については、P.A3-36「*メモリバリア*」を参照して下さい。

メモリコヒーレンシ
メモリ位置が読み出された（データ読み出しまたは命令フェッチにより）とき、実際に読み出される値が、その位置に最後に書き込まれた値と常に同じであることを保証する問題。物理的なメモリがメインメモリ、ライトバッファ、キャッシュなど複数の場所に分かれている場合、この問題の解決は簡単ではないことがあります。

メモリヒント
メモリヒント命令を使用すれば、データをレジスタファイルとの間で実際にロード / ストアしなくても、将来のメモリアクセスに関する事前情報をメモリシステムに提供することができます。ARMv7-M で定義されているメモリヒント命令は PLD と PLI のみです。

メモリマップド I/O
I/O 機能に対してロードやストアを行うとき、特別なメモリアドレスを使用する I/O 方式。

メモリ保護ユニット (MPU)
レジスタを使用してメモリ内に限られた数の保護領域を設定し、単純な制御を行うハードウェアユニット。

MPU メモリ保護ユニット参照。

NRZ 0 に復帰しない (Non-Return-to-Zero) の略。非同期通信ポートで使用される物理レイヤ信号方式です

マルチルコピーのアトミック性
P.A3-22「*マルチコピーのアトミック性*」で説明されているアトミック性の形式。

アトミック性、シングルコピーのアトミック性も参照。

オフセットアドレシング
メモリアドレスが、ベースレジスタの値にオフセットを加算または減算して計算されることを意味します。

物理アドレス
メインメモリ上の位置を示します。

ポストインデクスアドレシング
メモリアドレスとしてベースレジスタの値を使用しますが、ベースレジスタの値にオフセットを加算または減算して、結果をベースレジスタへ書き戻すことを意味します。

プリフェッチ
前の命令の実行が完了する前に、メモリから命令をフェッチする処理。プリフェッチされた命令は、必ず実行されるとは限りません。

プリインデクスアドレシング
メモリアドレスをオフセットアドレシングと同じ方法で計算するが、そのメモリアドレスをベースレジスタへ書き戻すことを意味します。

特権アクセス
メモリシステムは通常、より制限の多いユーザアクセスより、スーパーバイザアクセス許可に違反する特権モードからのメモリアクセスをチェックします。また一部の命令は、特権モードでのみ使用できます。

保護領域
メモリ保護ユニットのレジスタによって位置、サイズ、その他のプロパティが定義されているメモリ領域。

保護ユニット
メモリ保護ユニット参照。

擬似命令
ある命令エンコードにアセンブルされるが、それは別の異なるアセンブラ構文に逆アセンブルされることが期待される UAL アセンブラ構文で、本書ではその異なる構文で説明されています。例えば、MOV <Rd>,<Rm>, LSL #<n> は擬似命令で、LSL <Rd>,<Rm>,#<n> として逆アセンブルされることを前提としています。

PSR プログラムステータス レジスタ。APSR、EPSR、IPSR、xPSR *参照*。

RAZ 読み出し値 0 フィールド参照。

RAO/SBOP フィールド
読み出し値 1、書き込み時には 1 または保持。

このビットはどの実装でも 1（ビットフィールドの場合はすべてのビットが 1）として読み出され、フィールドへの書き込みは無視される必要があります。

ソフトウェアは、このフィールドからの読み出し値が 1（またはすべて 1）であることを前提とできますが、書き込み時には SBOP ポリシーを守る必要があります。

RAZ/SBZP フィールド
読み出し値 0、書き込み時には 0 または保持。

このビットはどの実装でも 0（ビットフィールドの場合はすべてのビットが 0）として読み出され、このフィールドへの書き込みは無視される必要があります。

ソフトウェアは、このフィールドからの読み出し値が 0 であることを前提とできますが、書き込み時には SBZP ポリシーを守る必要があります。

読み出し値 0 フィールド (RAZ)
読み出した値が常に 0 になります。

読み出し - 変更 - 書き込みフィールド (RMW)
汎用レジスタを読み出し、レジスタで関連フィールドを更新し、レジスタの値を書き戻す操作を行う必要があるフィールド。

予約
予約されているレジスタと命令は、特記されていない限り予測不能です。予約と記載されているビット位置は UNK/SBZP です。

復帰リンク
復帰アドレスに関連する値。

R/W1C R/W1C にマークされているレジスタのビットは、通常に読み出すことができ、"1 を書き込んでクリア " をサポートしています。レジスタを読み出してから結果をレジスタへ書き戻すと、セットされているすべてのビットがクリアされます。R/W1C は読み出し - 変更 - 書き込み時に、レジスタの読み出しと値の書き戻しとの間でビットがセットされた場合にエラーが発生することを防止します（そのビットは 0 として書き込まれるため、クリアされません）。

RAZ/WI レジスタまたはレジスタのフィールドに関する用語。読み出し値は 0 で、書き込みは無視されます。RAZ は単独で使用できます。

RO 読み出し専用のレジスタ、またはレジスタのフィールド。RO ビットへの書き込みアクセスは無視されます。

RISC 縮小命令セットコンピュータの略。

RMW 読み出し - 変更 - 書き込みフィールド参照。

丸めエラー
数値演算の丸め結果と、演算の正確な結果との差と定義されます。

飽和演算
表現可能な最大の数値より大きな結果を表現可能な最大の数値に設定し、表現可能な最小の数値より小さな結果を表現可能な最小の数値に設定する整数演算。符号付きの飽和演算は、DPS アルゴリズムで多く使用されます。これに対して、ARM プロセッサで使用される通常の符号付き整数演算では、オーバフローした結果は $+2^{31} - 1$ から -2^{31} へ、またはその逆にラップアラウンドされます。

SBO 常に 1 フィールド参照。

SBOP 常に 1 または保持フィールド参照。

SBZ 常に 0 フィールド参照。

SBZP 常に 0 または保持フィールド参照。

SCB システム制御ブロックの略。システム制御空間内のアドレス領域で、主要な機能の制御と、例外モデルに関連する構成に使用されます。

SCS システム制御空間の略。メモリマップでシステムの制御および構成用に予約されている 4KB の領域。

セキュリティホール
システムの保護をバイパスする機構。

セット レジスタまたはレジスタのフィールドに関連した用語。特記されていない限り、ビットの値が 1 であること（またはビットフィールドの値がすべて 1 であること）、あるいは 1 またはすべて 1 を書き込むことを意味します。

SWO シリアルワイヤ出力の略。NRZ、Manchester、または両方のエンコードをサポートする非同期 TPIU ポートです。

SWV シリアルワイヤ ビューアの略。SWO および DWT/ITM データトレース機能を組み合わせたものです。

自己変更コード
1 つまたは複数の命令をメモリに書き込んでから、それらの命令を実行するコード。このタイプのコードは、同期を保証するためのバリア命令を使用しなければ信頼性がありません。

常に 1 フィールド (SBO)
ソフトウェアで 1（ビットフィールドの場合はすべてのビットが 1）を書き込む必要があります。1 以外の値を書き込んだ場合の結果は予測不能です。

常に 1 または保持フィールド (SBOP)
以前に読み出しを行わずに値を書き込む場合、またはレジスタが初期化されていない場合は、ソフトウェアで 1（ビットフィールドの場合はすべてのビットが 1）を書き込む必要があります。レジスタを以前に読み出している場合、同じプロセッサの同じフィールドから以前に読み出したものと同じ値を書き込み、フィールドの値を保持する必要があります。

ハードウェアは、これらのフィールドへの書き込みを無視する必要があります。

1（ビットフィールドの場合はすべてのビットが 1）でもなく、同じプロセッサの同じフィールドから以前に読み出した値でもない値をこのフィールドへ書き込んだ場合、結果は予測不能です。

常に 0 フィールド (SBZ)
ソフトウェアで 0（ビットフィールドの場合はすべてのビットが 0）を書き込む必要があります。0 以外の値を書き込んだ場合の結果は予測不能です。

常に 0 または保持フィールド (SBZP)
以前に読み出しを行わずに値を書き込む場合、またはレジスタが初期化されていない場合は、ソフトウェアで 0（ビットフィールドの場合はすべてのビットが 0）を書き込む必要があります。レジスタを以前に読み出している場合、同じプロセッサの同じフィールドから以前に読み出したものと同じ値を書き込み、フィールドの値を保持する必要があります。

ハードウェアは、これらのフィールドへの書き込みを無視する必要があります。

0（ビットフィールドの場合はすべてのビットが 0）でもなく、同じプロセッサの同じフィールドから以前に読み出した値でもない値をこのフィールドへ書き込んだ場合、結果は予測不能です。

符号付きデータタイプ
$-2^{N-1} \sim +2^{N-1}-1$ の範囲で、2 の補数形式を使用して表現される整数。

符号付きイミディエートおよびオフセットフィールド
特記されていない限り、2 の補数表記でエンコードされます。

SIMD 単一命令、複数データ操作の意味。

シングルコピーのアトミック性
P.A3-21「*シングルコピーのアトミック性*」で説明されているアトミック性の形式。

アトミック性、マルチコピーのアトミック性も参照。

空間的局所性
プログラムがメモリ位置へアクセスした後で、近くのメモリ位置にも直後にアクセスする可能性が高いという考察結果。マルチワード キャッシュラインを持つキャッシュは、この考察を利用してパフォーマンスを向上しています。

サブアーキテクチャ定義
動作がサブアーキテクチャの定義で指定されることを期待していることを意味します。一般に、この定義は複数の実装で共有されますが、特定のタイプのコードのみがこの定義を前提とする必要があります。これによって、新しいサブアーキテクチャの開発が必要なとき、ソフトウェアの変更の必要が最小限になります。

SVC スーパバイザコールの略。

SWI SVC の意味で以前に使用されていた用語。

ステータスレジスタ
APSR、EPSR、IPSR、xPSR *参照*。

時間的局所性
プログラムがメモリ位置へアクセスした後で、近いうちに同じメモリ位置へ再度アクセスする可能性が高いという考察結果。キャッシュはこの考察を利用してパフォーマンスを向上しています。

Thumb 命令
Thumb 状態のプロセッサが実行する動作を指定する、1 つまたは 2 つのハーフワード。Thumb 命令は、ハーフワードアラインドの必要があります。

Thumb-2
Thumb 状態での 16 ビットおよび 32 ビットのアトミックな命令の実行サポート。

TPIU トレースポート インタフェースユニットの略。ARM デバッグアーキテクチャの一部です。

UAL 統一アセンブラ言語参照。

アンアラインド
アンアラインドアクセスは、アクセスのアドレスがアクセスする要素のサイズにアラインしていないアクセスを指します。

アンアラインドメモリアクセス
適切にハーフワード、ワード、またはダブルワードにアラインしていない、またはその可能性があるメモリアクセス。

未割り当て
特記されていない限り、アーキテクチャが命令のビットパターン全体に特定の機能を割り当てておらず、代わりに未定義、予測不能、または未割り当てのヒント命令と記述されている場合、その命令エンコードは未割り当てです。

レジスタのビットについては、アーキテクチャでそのビットに機能が割り当てられていない場合、そのビットは未割り当てです。

未定義
未定義命令例外を生成する命令を指します。

統一アセンブラ言語
Thumb-2 で導入されたアセンブラ言語で、本書で使用されています。詳細については、P.A4-4「*統一アセンブラ言語*」を参照して下さい。

統一キャッシュ
命令フェッチとデータのロード / ストア、両方の処理に使用されるキャッシュ。

インデクスなしのアドレシング
ベースレジスタの値を直接アドレスとしてメモリへ送信し、オフセットの加減算を行わないアドレシング。ほとんどのタイプのロード / ストア命令では、オフセットアドレシングのイミディエートオフセットに 0 を指定することで、インデクスなしのアドレシングを行います。LDC、LDC2、STC、STC2 命令には明示的なインデクスなしのアドレシングモードがあり、命令のオフセットフィールドを追加のコプロセッサオプションの指定に使用できます。

未知 未知の値には有効なデータが含まれておらず、時間ごと、命令ごと、実装ごとに変化する可能性があります。未知の値がセキュリティホールとなることは回避する必要があります。未知な値を文書化したり、ひとつの定義された値や効果を持つものとして取り扱ったりしてはいけません。

UNK/SBOP フィールド
読み出し値未知、書き込み時には 1 または保持。

このビットはどの実装でも 1（ビットフィールドの場合はすべてのビットが 1）として読み出され、フィールドへの書き込みは無視される必要があります。

ソフトウェアは、このフィールドからの読み出し値が 1（またはすべて 1）であることを前提としてはいけません。書き込み時には SBOP ポリシーを守る必要があります。

UNK/SBZP フィールド
読み出し値未知、書き込み時には 0 または保持。

このビットはどの実装でも 0（ビットフィールドの場合はすべてのビットが 0）として読み出され、フィールドへの書き込みは無視される必要があります。

ソフトウェアは、このフィールドからの読み出し値が 0 であることを前提としてはいけません。書き込み時には SBZP ポリシーを守る必要があります。

UNK フィールド
未知の値（UNKNOWN）が格納されています。

予測不能
動作を予測できないことを意味します。予測不能な動作がセキュリティホールとならないようにする必要があります。予測不能な動作によって、プロセッサまたはシステムのどの部分も停止、ハングアップすることは許されません。予測不能な振る舞いを文書化したり、ひとつの定義された効果を持つものとして取り扱ったりしてはいけません。

符号なしデータタイプ
0 ～ + 2^N–1 の範囲の負でない整数で、通常のバイナリ形式を使用します。

ウォッチポイント
メモリへのアクセスによってトリガされるデバッグイベントで、アクセスされるメモリのアドレスによって定義されます。

ワード 32 ビットのデータアイテム。ARM システムでのワードは、通常はワードアラインドです。

WO 書き込み専用のレジスタ、またはレジスタのフィールド。WO ビットの読み出しアクセス時の値は未知です。

ワードアラインド
アドレスが 4 の倍数であることを意味します。

ライトバッファ
高速なメモリのブロックで、メインメモリへのストアを最適化することを目的としています。

WYSIWYG
見たとおりの結果が得られる (What You See Is What You Get) の略で、生成される出力が予測される動作と同じであることを意味します。一般的には、画面表示と印刷されたページが同一であることや、ソフトウェアのソースと実行可能なコードが一致していることなどを指して使われます。

xPSR APSR、EPSR、IPSR を組み合わせた単一の 32 ビット プログラムステータス レジスタを指す用語。詳細については、P.B1-7「*専用プログラム ステータスレジスタ (xPSR)*」を参照して下さい。

